Esercizi su stringhe

```
// funzione che concatena due stringhe

void concatenastringhe (char st1[], char st2[], char stOut[])
{
  int n=0,s=0;       // dichiarazione variabili con inizializzazioni

  while (st1[n] != '\0')         // copia prima stringa tranne '\0'
    stOut[n++] = st1[n];

  do {                    // copia di seguito 2ª stringa compreso '\0'
    stOut[n++] = st2[s];  // notare che n parte dal valore assegnato
  } while (st2[s++] != '\0')    // dal while precedente (1ª stringa)
}
```


Esercizi su stringhe

```
// funzione che mischia due stringhe

void mischiastringhe (char st1[], char st2[], char stOut[])
{
  int n=0,s=0;

  // copia fintanto che ci sono caratteri su tutte e due le stringhe
  while ((st1[s] != '\0') && (st2[s] != '\0')) {
    stOut[n++] = st1[s];       // copia il carattere dalla 1ª stringa
    stOut[n++] = st2[s++];     // copia il carattere dalla 2ª stringa
  }
  if(st1[s] != '\0')  // se ci sono ancora caratteri nella 1ª stringa
     do {                                 // copia i restanti caratteri
       stOut[n++] = st1[s];              // della 1ª stringa in stOut[]
     } while (st1[s++] != '\0')           // incluso il carattere '\0'
   else    // altrimenti ci possono essere caratteri nella 2ª stringa
     do {            // quindi copia gli eventuali restanti caratteri
       stOut[n++] = st2[s];              // della 2ª stringa in stOut[]
     } while (st2[s++] != '\0')           // incluso il carattere '\0'
}
```

Esercizi su stringhe

- Esercizio 6:

  Scrivere una funzione che date due stringhe (di lunghezza massima M) restituisca 1 se le stringhe sono uguali, 0 altrimenti:

```
int comparastringhe(char st1[], char st2[])
```


Esercizi su stringhe

```
// funzione che compara due stringhe

int comparastringhe (char st1[], char st2[])
{
  int n=0;                                  // dichiarazione variabile con inizializzazioni
   while (st1[n] == st2[n] )                 // finché ci sono caratteri uguali vado avanti
    if(st1[n++] == '\0') return 1;           // esco con 1 se raggiungo il carattere nullo
                                             // notare che poiché sono nel corpo del while
                                             // st1[n] == st2[n] == '\0')
   return 0;  // se sono arrivato qui è perché st1[n] != st2[n] e quindi ritorno 0 (falso)
}
```

Strutture

- Una struttura (**struct**) è un insieme di tipi diversi di dati raggruppati in un'unica dichiarazione. La forma della dichiarazione è la seguente:

```
struct nome_modello {
  tipo1 nome_elemento1;
  tipo2 nome_elemento2;
  tipo3 nome_elemento3;
        .
        .
} nome_oggetto;
```

in cui ***nome_modello*** è un nome per il modello di struttura e ***nome_oggetto*** (opzionale) è un identificatore che denota un oggetto avente la struttura ***nome_modello*** .

Tra le parentesi graffe **{}** sono indicati i tipi e i rispettivi sub_identificatori degli elementi che compongono la struttura.

Strutture

- Una volta dichiarata una struttura il suo nome può essere utilizzato come un nuovo tipo di dati al pari di quelli fondamentali come **int** , **char** o **short**. Ad esempio:

```
struct prodotti {
  char nome [30];
  float prezzo;
} ;
prodotti mele;
prodotti arance, meloni;
```

definizione modello di struttura **prodotti** con due campi: nome e prezzo, di tipi diversi.

dichiarazione di tre oggetti di tipo **prodotti**: mele , arance e meloni.

- Il campo opzionale **nome_oggetto** che compare alla fine della dichiarazione di una struttura serve a dichiarare direttamente oggetti di tale tipo:

```
struct prodotti {
  char nome [30];
  float prezzo;
} mele, arance, meloni;
```

Definizione struttura e dichiarazioni variabili di tipo **prodotti** equivalente alla precedente.

  - In questo caso, in cui inseriamo direttamente nella dichiarazione della struttura la dichiarazione di tutti gli oggetti di tale tipo, il nome **nome_modello** (**prodotti** nel nostro caso) è opzionale.
  - Se **nome_modello** viene omesso non possiamo però dichiarare in seguito altri oggetti dello stesso tipo.

Strutture

- Occorre distinguere chiaramente tra i concetti di modello della struttura e di oggetto appartenente alla struttura.
- In analogia con i termini che abbiamo usato per le variabili possiamo dire che:
  - il modello è il tipo
  - l'oggetto è la variabile

Si possono dichiarare molti oggetti (variabili) di uno stesso modello (tipo).

- Una volta dichiarati i tre oggetti mele, arance e meloni possiamo operare sui campi che li costituiscono.
- Per fare questo bisogna usare un punto (.) tra il nome dell'oggetto ed il nome del campo. Ad esempio:

```
mele.nome
mele.prezzo
arance.nome
arance.prezzo
meloni.nome
meloni.prezzo
```

ciascuna di esse appartiene al rispettivo tipo:

- **`mele.nome , arance.nome`** e **`meloni.nome`** sono di tipo **`char[30]`**
- **`mele.prezzo , arance.prezzo`** e **`meloni.prezzo`** sono di tipo **`float`**

Strutture: esempio

```
// esempio con le strutture
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

struct film_t {
  char titolo [50];
  int anno;
} mio, tuo;

void stampaFilm (film_t film);

int main ()
{
  strcpy (mio.titolo, "2001 Odissea nello spazio");
  mio.anno = 1968;

  cout << "Dammi il titolo: ";
  cin.getline (tuo.titolo,50);
  cout << "Dammi l'anno: ";
  cin >> tuo.anno;

  cout << "Il mio film preferito e':\n";
  stampaFilm (mio);
  cout << "Il tuo e':\n";
  stampaFilm (tuo);
  return 0;
}

void stampaFilm (film_t film)
{
  cout << film.titolo << " (" << film.anno << ")\n";
}
```

```
Dammi il titolo: Alien
Dammi l'anno: 1979
Il mio film preferito e':
2001 Odissea nello spazio (1968)
Il tuo e':
Alien (1979)
```

Strutture: esempio

```
// array di strutture
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

#define N_FILM 5

struct film_t{
  char titolo [50];
  int anno;
} films[N_FILM];

void stampaFilm (film_t film);

int main ()
{
  int n;
  for (n=0; n<N_FILM; n++)
  {
    cout << "Dammi il titolo: ";
    cin.getline (films[n].titolo,50);
    cout << "Dammi l'anno: ";
    cin >> films[n].anno;
  }
  cout << "\nHai inserito i seguenti film:\n";
  for (n=0; n<N_FILM; n++)
    stampaFilm (films[n]);
  return 0;
}
void stampaFilm (film_t film)
{
  cout << film.titolo << " (" << film.anno << ")\n";
}
```

```
Dammi il titolo:  Alien
Dammi l'anno: 1979
Dammi il titolo: Blade Runner
Dammi l'anno: 1982
Dammi il titolo: Matrix
Dammi l'anno: 1999
Dammi il titolo: Rear Window
Dammi l'anno: 1954
Dammi il titolo: Taxi Driver
Dammi l'anno: 1975
Hai inserito i seguenti film:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)
```

Strutture annidate

- Le strutture si possono annidare in modo tale che un elemento di una struttura può essere a sua volta una struttura:

```
struct film_t {
  char titolo [50];
  int anno;
};

struct amici_t {
  char nome [50];
  char email [50];
  film_t film_preferito;
} amico1, amico2;
```

dopo tale dichiarazione possiamo usare le espressioni:

```
amico1.nome
amico2.film_preferito.titolo
amico1.film_preferito.anno
```

Altri tipi definiti dal programmatore

- Oltre alle strutture, il programmatore può definire altri tipi di dato:
  - definizione di propri tipi (**typedef**);
  - unioni (**union**);
  - enumerazioni (**enum**).

- In particolare:
  - **typedef** permette di definire nuovi tipi basati su altri tipi di base o precedentemente definiti;
  - **union** permette di utilizzare la stessa zona di memoria per memorizzare oggetti appartenenti a tipi differenti;
  - **enum** permette di creare tipi non basati su tipi precedentemente definiti.

typedef

- **typedef** si usa:
  - per definire un tipo che viene usato molte volte nel programma e che è possibile debba essere cambiato in una nuova versione del programma.
  - per dare un nome sintetico quando il tipo ha un nome molto complesso.
- Per fare questo si usa la parola chiave **typedef**, nel seguente modo:

```
typedef   tipo_esistente   nome_nuovo_tipo ;
```

in cui ***tipo_esistente*** è un tipo fondamentale del C++ o un altro tipo definito precedentemente e ***nome_nuovo_tipo*** è il nome assegnato al nuovo tipo definito.

- Ad esempio:

```
typedef char C;              // definisce il tipo C come char
typedef unsigned int WORD;   // definisce il tipo WORD come unsigned int
typedef char[50] campo;      // definisce il tipo campo come char[50]
```

```
I nuovi tipi si possono usare in seguito scrivendo, ad esempio:
   C unchar, altrochar;
   WORD parola;
   campo nome;
```

union

- **La dichiarazione di union** è simile a quella vista per le strutture, ma ha significato completamente diverso:

```
union nome_modello {
    tipo1 elemento1;
    tipo2 elemento2;
    tipo3 elemento3;
    .
    .
  } nome_oggetto;
```

tutti gli elementi della **union** occupano lo stesso spazio di memoria. La dimensione di tale spazio è quella dell'elemento che ne richiede di più.

- Ad esempio:

```
union tipi_t {
 char c;
 int i;
 float f;
 } x;
```

definisce tre elementi: **x.c, x.i, x.f** ciascuno di un tipo diverso.
Essi non si possono usare contemporaneamente in quanto utilizzano la stessa memoria:

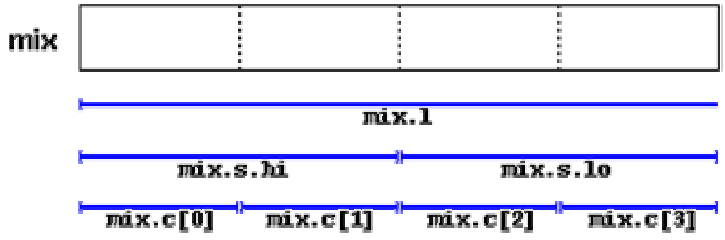
- l'oggetto **x** può contenere un valore di tipo char o un valore di tipo int o un valore di tipo float ma soltanto uno di essi.

union

- Un possibile uso di una **union** è l'unione di un tipo elementare con una struttura o un array di elementi più piccoli. Ad esempio:

```
union mix_t{
    long l;
    struct {
      short hi;
      short lo;
      } s;
    char c[4];
  } mix;
```

definisce tre modi di accedere allo stesso gruppo di 4 byte : **mix.l, mix.s** e mix.c.
Ecco i diversi modi di accedere:



Unioni anonime

- In C++ possiamo avere delle unioni anonime.
- Se inseriamo una unione in una struttura senza indicare il nome di un oggetto (il nome che viene di norma posto alla fine, dopo la parentesi graffa }) l'unione si dice anonima e possiamo accedere direttamente agli elementi come fossero campi della struttura.
- Osserviamo la differenza tra le seguenti due dichiarazioni:

unione

```
struct {
  char titolo[50];
  char autore[50];
  union {
    float euro;
    int lire;
  } prezzo;
} libro;
```

unione anonima

```
struct {
  char titolo[50];
  char autore[50];
  union {
    float euro;
    int lire;
  };
} libro;
```

La differenza è che nel primo caso abbiamo dato un nome (**prezzo**) all'unione mentre nel secondo caso lo abbiamo omesso. Cambia modo di accedere ai valori **euro** e **lire**.
Nel primo caso si deve usare:

```
libro.prezzo.euro
libro.prezzo.lire
```

mentre nel secondo basta scrivere:

```
libro.euro
libro.lire
```

Ricordiamo che siccome si tratta di una unione i campi euro e lire occupano la stessa zona di memoria e quindi non si possono usare per memorizzare due valori diversi: si può registrare il prezzo in euro oppure in lire ma non entrambi.

enum

- Le enumerazioni permettono di creare dei nuovi tipi che non sono basati sui tipi definiti precedentemente. La forma della dichiarazione è la seguente:

```
enum nome_enum {
    id_valore1,
    id_valore2,
    id_valore3,
    .
    .
  } nome_oggetto;
```

- Possiamo, ad esempio, creare un nuovo tipo **color** per memorizzare dei colori:

```
enum colori {nero, blu, verde, viola, rosso, porpora, giallo, bianco};
```

- Notiamo che nella dichiarazione non abbiamo usato nessun tipo di dato fondamentale. Abbiamo creato un nuovo tipo di dato non basato su altri esistenti: il tipo **colori**, i cui possibili valori sono tutti e soli gli identificatori che abbiamo racchiuso tra parentesi graffe {}.

enum

- Dopo aver dichiarato il tipo enumerazione **colori** possiamo scrivere:

```
colori c;

c = blu;
if (c == verde) c = rosso;
```

- In realtà il compilatore rappresenta gli elementi di un tipo enumerazione usando degli interi.
- Se non specificato altrimenti il primo elemento è rappresentato con 0 ed i successivi con 1,2,... in successione.
- Vi è una conversione implicita da tipo enum a int mentre in senso inverso occorre indicare esplicitamente la conversione di tipo.
- Ad esempio per calcolare il colore successivo possiamo scrivere

```
enum c = colori(c+1);
```


Esercizi su strutture

- Esercizio 7:

  Scrivere un programma per l'inserimento e visualizzazione di un gruppo di automobili (massimo M) descritte da marca, modello e numero di unità vendute.

  Il gruppo di automobili è inserito dall'utente del programma e la visualizzazione avviene successivamente all'inserimento e tramite il nome fornito di una marca di cui si vogliono visualizzare le informazioni.