Capitolo 4

Strutture dati

4.1 Tipi strutturati

Nei capitoli precedenti abbiamo considerato soltanto tipi primitivi semplici, precisamente i tipi `int`, `float`, `char` e `bool` del C++. Ricordiamo che il tipo di una variabile indica l'insieme dei possibili valori che essa può assumere e l'insieme delle possibili operazioni applicabili a tali valori. Nel caso dei tipi semplici, il valore corrente di una variabile è un valore "atomico", non ulteriormente scomponibile.

In C++ (come nella maggior parte dei linguaggi di programmazione) sono presenti anche tipi non semplici, detti *tipi strutturati*, i cui valori sono costituiti ciascuno da un *insieme di valori*, a loro volta di tipo semplice o strutturato, omogenei o disomogenei, organizzati tra loro secondo opportuni criteri.

Tipo strutturato. Un *tipo strutturato* è un tipo di dato costituito da:

- il tipo, ed eventualmente il numero, dei suoi componenti;
- un criterio di aggregazione dei suoi componenti;
- l'insieme delle possibili operazioni applicabili all'intera struttura, tra cui una o più *operazioni di selezione* dei singoli elementi. □

I linguaggi di programmazione permettono, solitamente, di definire tipi strutturati nuovi a partire dai tipi (già esistenti) degli elementi che li costituiscono tramite opportuni *costruttori di tipo*. In particolare, il C++ offre i seguenti costruttori per la creazione di tipi strutturati:

- array (che in C++ non ha una parola chiave specifica, ma è implicito nella forma sintattica della dichiarazione)
- `struct`
- `union`
- `class`.

Costruttori diversi realizzano forme diverse di aggregazione dei dati e quindi tipi strutturati diversi, con proprietà diverse ed un diverso insieme di operazioni ad essi applicabili. Ad esempio, due tipi strutturati costituiti entrambi da tre elementi di tipo intero, ma definiti uno tramite il costruttore array e l'altro tramite il costruttore `struct` sono tipi distinti. Inoltre, tipi strutturati definiti tramite lo stesso costruttore possono lo stesso differire tra loro a causa del tipo e/o del numero dei componenti. Ad esempio, due tipi strutturati realizzati entrambi tramite il costruttore array, ma uno composto da tre elementi interi e l'altro da tre elementi reali (`float` in C++) sono tipi distinti.

Analizzeremo gli array nel sottocapitolo 4.3. Il costruttore `struct` sarà invece trattato nel sottocapitolo 4.6, mentre il costruttore `class` sarà introdotto ed analizzato in dettaglio nella II Parte del testo. Non tratteremo invece il costruttore `union` per il quale rimandiamo ad un manuale di linguaggio C, come ad esempio il classico [13].

4.2 Strutture dati astratte e concrete

Chiamiamo *struttura dati astratta* una struttura dati definita (soltanto) in base alle sue proprietà ed alle operazioni applicabili ad essa. Viceversa, l'implementazione, ossia quale *struttura dati concreta* è utilizzata per la sua realizzazione, non è visibile.

Esempi di tipiche strutture dati astratte sono:

- matrici (a k dimensioni)
- stringhe
- tabelle
- pile
- code
- insiemi
- alberi
- grafi.

Ciascuna di queste strutture dati ha precise caratteristiche che la distinguono dalle altre. Ad esempio, una *pila di interi* è costituita da una sequenza (anche vuota) di elementi di tipo intero su cui è possibile effettuare (soltanto) le operazioni di inserimento o di estrazione dell'elemento che si trova su uno dei due estremi della sequenza (la "testa" della pila). Una *coda di interi*, invece, è analoga alla pila di interi, ma le operazioni di inserimento o di estrazione di un elemento si effettuano una su un estremo della sequenza e l'altra sull'altro estremo.

Una stessa struttura dati astratta può essere implementata utilizzando diverse strutture dati concrete. Ad esempio, una pila può essere realizzata

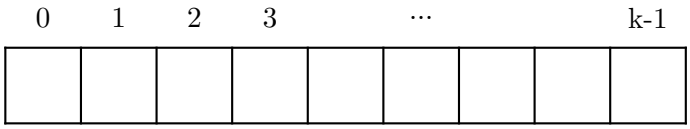
tramite un array ed una variabile intera che indica la posizione della "testa" della pila, oppure da una lista di elementi concatenati tramite puntatori a partire dall'elemento di testa. Viceversa, una stessa struttura dati concreta può essere utilizzata per implementare diverse strutture dati astratte. Per esempio gli array possono essere usati per realizzare matrici, stringhe, tabelle, pile, ecc..

In questo testo tratteremo soltanto alcune delle strutture dati (astratte) sopra elencate. Precisamente, in questo capitolo, mostreremo la definizione ed una possibile implementazione delle strutture dati matrice, stringa e tabella. Alcune altre strutture dati, tra cui le pile, saranno invece trattate nella II Parte del testo, dopo aver introdotto il costruttore `class`. Inoltre ritorneremo sulla nozione di struttura dati astratta in un capitolo successivo. Per una trattazione più completa ed approfondita delle diverse strutture dati e dei relativi algoritmi per la loro gestione rimandiamo, ad esempio, al testo [2].

4.3 Array

Array. Un oggetto di tipo *array* (o, più semplicemente, un *array*) è una sequenza di k elementi, $k \geq 0$, tutti di tipo t, individuati singolarmente da un numero intero compreso tra 0 e $k-1$ detto *indice*. k costituisce la *dimensione*, o *capacità*, dell'array. □

Una tipica rappresentazione grafica per un array di dimensione k è la seguente:



Nei paragrafi successivi esamineremo in dettaglio le possibilità per la creazione e manipolazione di array offerte dal C++. Qui ci limiteremo a considerare array statici e semi-dinamici, ovvero array creati tramite opportuna dichiarazione di array. Affronteremo invece la creazione di array dinamici in un capitolo successivo, dopo aver introdotto le nozioni di puntatore ed allocazione dinamica della memoria.

4.3.1 Dichiarazione di array in C++

Un array in C++ può essere creato tramite la seguente dichiarazione.

Dichiarazione di array. La dichiarazione

$$t \ \texttt{A[}k\texttt{];}$$

dove:

- `A` è un identificatore,
- t è un tipo qualsiasi (semplice o a sua volta strutturato),
- k è un'espressione intera ≥ 0,

è una *dichiarazione di array* in C++. Il significato di questa dichiarazione è la creazione di una variabile di nome `A`, di tipo strutturato *array di* t (ovvero t `[ ]`), costituita da k elementi di tipo t. Più sinteticamente, diremo che la dichiarazione t `A[`k`]` crea un array `A` di k elementi di tipo t. □

Come per le variabili di tipo semplice, l'elaborazione di una dichiarazione di array comporta l'allocazione dello spazio di memoria necessario a contenere gli elementi dell'intero array, ovvero, nel caso della dichiarazione t `A[`k`]`, l'allocazione di k celle di memoria consecutive, ciascuna di dimensioni adeguate a contenere un valore di tipo t. Nel seguito ci riferamo principalmente ad esempi in cui k sia costante (array statici), ma le considerazioni fatte valgono in generale. Alcune osservazioni specifiche sul caso di k variabile (array semi-dinamici) verranno fornite nel paragrafo 4.3.6.

Si osservi che nella dichiarazione t `A[`k`]` il tipo della variabile `A` (array di t) è definito direttamente nella dichiarazione stessa. Si osservi anche la sintassi piuttosto "infelice" della dichiarazione di array in C++, in cui il nome della variabile `A` appare in mezzo a quello del suo tipo (t `[ ]`) e in quest'ultimo non è presente alcuna parola chiave che indichi la presenza del costruttore di array (come invece accade, ad esempio, in Pascal). In un capitolo successivo (Capitolo 4.8.1) vedremo come in C++ sia anche possibile dichiarare il tipo array a parte, associandogli un nome, da usare poi nella dichiarazione di array.

Esempio 4.1 (Dichiarazioni di array)

`int V1[10];`	*variabile di nome* `V1`*,*
	di tipo array di interi, di dimensione 10
`int V2[20];`	*variabile di nome* `V2`*,*
	di tipo array di interi, di dimensione 20
	(n.b. stesso tipo dell'array `V1`*)*
`char S[32];`	*variabile di nome* `S`*,*
	di tipo array di caratteri, di dimensione 32

Nel seguito spesso scriveremo, per comodità, "l'array di interi `A`" invece che "la variabile di nome `A` di tipo array di interi".

In una dichiarazione di array, come nella dichiarazione di variabili semplici, è possibile specificare anche un eventuale valore iniziale che, in questo caso, sarà costituito da un insieme di valori da assegnare agli elementi dell'array.

Dichiarazione di array con inizializzazione. La dichiarazione

$$t\ \texttt{A[}k\texttt{]} = \{i_1, i_2, \ldots, i_m\};$$

dove i_1,i_2,...,i_m sono espressioni di tipo t, è una *dichiarazione di array con inizializzazione* (forma base). $\{i_1, i_2, \ldots, i_m\}$ è detta *lista di inizializzatori*. Il significato di questa dichiarazione è la creazione di un array di dimensione k, con l'inizializzazione dei suoi primi m elementi, rispettivamente, con i valori di i_1,i_2,...,i_m. □

Esempio 4.2 (Dichiarazione di array con inizializzazione)

```
char vocali[5] = {'a','e','i','o','u'};
```

array di 5 caratteri, di nome `vocali`*, inizializzati nel modo seguente:*

0	1	2	3	4
'a'	'e'	'i'	'o'	'u'

Approfondimento (Lista parziale di inizializzatori). Se il numero degli elementi specificati nella lista di inizializzatori è maggiore della capacità dell'array, cioè $m > k$, si verifica un errore rilevato in fase di compilazione; se invece è $m < k$ allora vengono inizializzati i primi m elementi dell'array, mentre i restanti $k-m$ elementi vengono riempiti con 0, indipendentemente dal loro tipo.

Esempio 4.3 (Dichiarazione di array con inizializzazione parziale)

```
int V[10] = {2,-1,1};
```

array di 10 interi di cui i primi 3 inizializzati con i valori `2`*,* `-1` *e* `1`*, rispettivamente, ed i rimanenti elementi tutti inizializzati con* `0`*.*

Si osservi che, come per le variabili di tipo semplice, nel caso in cui non si specifichi alcun valore iniziale, gli elementi di un array avranno un valore *indeterminato*.

La presenza di una lista di inizializzatori permette di dichiarare l'array senza dover necessariamente indicare in modo esplicito la sua dimensione. In questo caso, infatti, la dimensione dell'array è "dedotta" automaticamente dal numero di inizializzatori presenti.

Esempio 4.4 (Dichiarazione di array con dimensione implicita)

```
float F[] = {1.0,-1.5,+1.5};
```

array di 3 elementi di tipo `float`*.*

Nota. Si presti attenzione al fatto che la dichiarazione di un array in cui non si specifichino dimensioni, in assenza di una inizializzazione esplicita, equivale a creare un array di dimensioni 0, ovvero un array senza lo spazio di memoria necessario a contenere eventuali elementi.

Si noti che l'assegnazione di valori agli elementi di un'array tramite una lista di inizializzatori è possibile soltanto all'interno della dichiarazione dell'array stesso. Non è quindi possibile scrivere, ad esempio:

```
int A[5];
A = {1, 2, 3, 4, 5};
```

che comporterebbe una segnalazione di errore da parte del compilatore.

4.3.2 Operazione di selezione

Operazione fondamentale per poter utilizzare un array, e più in generale una qualsiasi struttura dati, è quella di poter riferirsi, e quindi accedere, sia in lettura che in scrittura, ad un suo singolo elemento. Le caratteristiche di questa operazione, detta *operazione di selezione*, differiscono da struttura a struttura. Nel caso dell'array l'operazione di selezione di un elemento avviene semplicemente tramite l'indice dell'elemento stesso, ovvero il numero intero associato univocamente ad ogni elemento dell'array. Si tratta di un *accesso diretto*, il cui tempo d'accesso non dipende dalla posizione relativa dell'elemento all'interno dell'array, ma è costante (in contrasto con l'*accesso sequenziale*). L'implementazione dell'array fornita dal linguaggio dovrà garantire questa importante proprieà.

In C++ la selezione di un elemento di un array avviene tramite un'opportuna espressione.

Espressione con indice. Dato l'array

$$t \ \texttt{A[}k\texttt{];}$$

con t tipo qualsiasi, l'espressione

$$\texttt{A[}e\texttt{]}$$

dove e è un'espressione di tipo compatibile con `int`, è detta *espressione con indice*. Il significato di `A[`e`]` è quello di selezionare l'elemento dell'array `A` di indice il valore dell'espressione e (si noti che prima si valuta e e quindi si usa il valore risultate per selezionare l'elemento dell'array). Il tipo dell'espressione `A[`e`]` è t, e cioè il tipo degli elementi di `A`. □

Esempio 4.5 (Selezione di elementi di un array) *Dato l'array di interi*

```
int V[10] = {5,11,20,17,8,4,9,13,5,12};
```

e cioè, graficamente,
le seguenti sono operazioni corrette di accesso in lettura a singoli elementi dell'array:

0	1	2	3	4	5	6	7	8	9
5	11	20	17	8	4	9	13	5	12

```
cout << V[0];    // stampa il valore 5
int i = 2;
cout << V[i];    // stampa il valore 20
cout << V[i+1];  // stampa il valore 17
cout << V[i]+1;  // stampa il valore 21
cout << V[2];    // stampa il valore 20
```

Negli esempi di sopra l'espressione `V[...]` denota il *valore* dell'elemento di `V` selezionato. L'espressione `V[...]` può comparire però anche nella parte sinistra di un'istruzione di assegnamento ed in questo caso denota la *locazione* dell'elemento selezionato, non il suo valore. L'istruzione `V[i] =` *expr* permette perciò di assegnare il valore dell'espressione *expr* alla locazione di memoria individuata dall'espressione `V[i]`, e cioè all'elemento dell'array `V` di indice i.

Esempio 4.6 (Accesso in lettura e scrittura ad elementi di un array) *Sia* `V` *l'array dell'Esempio 4.5.*

```
int i = 2;
V[1] = 7;
V[2] = V[1] * V[0];
V[i] = V[i] + 1;    // equivalente a V[i]++
V[i] = V[i+1];
int W[5] = {7, 3, 1, 4, 2}
cout << V[W[i]];    // stampa il valore 7
```

Il singolo elemento di un array `A` con elementi di tipo t, individuato tramite l'operazione di selezione `A[`e`]`, è a tutti gli effetti una variabile di tipo t. Dunque è possibile applicare ad esso tutte le operazioni previste dal tipo t. Ad esempio, se `V` è l'array di interi dell'Esempio 4.5 possiamo scrivere ad esempio

```
int x = V[1] % 2;  //resto divisione intera tra V[1] e 2
```

oppure

```
V[2] = V[1] * V[0];  //moltiplicazione tra interi
```

L'array è una collezione limitata di valori di un dato tipo t, il cui numero massimo è stabilito dal valore di k specificato nella dichiarazione dell'array

stesso. Di conseguenza, gli indici di un array possono assumere valori in un intervallo ben definito, che in C++ è costituito dagli interi tra 0 e $k-1$. Qualsiasi riferimento ad un elemento dell'array con un indice di valore minore di 0 o maggiore o uguale a k è da considerarsi un errore, che produce un risultato indefinito.

In C++ però è stata fatta la scelta, essenzialmente per motivi di efficienza, di non controllare, nè a compile-time, nè a run-time, la correttezza (dei valori) degli indici di un array. Dunque in C++ è possibile scrivere ad esempio

```
int V[10];
V[11] = 3;
```

senza che il compilatore dia nessuna segnalazione di errore. L'esecuzione dell'operazione di selezione sull'array comporterà un accesso ad una locazione di memoria che non fa parte dello spazio di memoria allocato per l'array e quindi il risultato del programma diventa del tutto imprevedibile (ad esempio, potrebbe andare a scrivere il valore da assegnare all'elemento dell'array nella locazione di memoria di un'altra variabile del programma, oppure, se l'indice usato è molto più grande del massimo previsto, provocare una violazione di indirizzamento della memoria principale con conseguente terminazione forzata dell'esecuzione del programma). Si tratta quindi di errori, in generale, molto insidiosi e difficili da individuare. È dunque importante prestare molta attenzione alla correttezza dei valori attribuiti agli indici di un array[1].

Memorizzazione di un array

La creazione di un array `A` di k elementi di tipo t (e cioè, t `A[`k`]`) comporta l'allocazione di k celle di memoria di tipo t, consecutive, in cui memorizzare nell'ordine i k elementi dell'array. Per *cella di memoria* di tipo t si intende la sequenza di m byte consecutivi utilizzati per memorizzare un dato di tipo t. Ad esempio, per `int`, sarà, tipicamente, $m = 4$, mentre per `char` sarà $m = 1$. L'*indirizzo base* b dell'area di memoria dedicata all'array viene stabilito dal sistema al momento dell'allocazione dell'array stesso e l'utente non ne ha nessun controllo.

Esempio 4.7 (Allocazione di un array) *L'array*

```
int V[6];
```

comporta l'allocazione di 6 *celle di tipo intero e quindi occupa* $6 \times 4 = 24$ *byte. Se assumiamo* $b = 15216$ *e* $m = 4$ *la situazione della memoria dedicata all'array* `V` *è la seguente:*

[1] In C++ è possibile usare in alternativa al costruttore primitivo di array la classe predefinita `vector` della libreria standard o definire una propria classe che preveda anche il controllo a run-time di correttezza degli indici.

$V[0]$	$b = 15216$
$V[1]$	15220
$V[2]$	15224
$V[3]$	15228
$V[4]$	15232
$V[5]$	15236

Si noti che, poichè gli indirizzi di memoria sono relativi al singolo byte, la cella successiva alla prima avrà indirizzo (in byte) $b + 4$ (nell'esempio, $15216 + 4$).

Con questa implementazione è facile garantire l'*accesso diretto* ai singoli elementi dell'array, cioè che il tempo di accesso al singolo elemento sia indipendente dalla sua posizione. Dato un array t `A[`k`]` con indirizzo base b e dimensione (in byte) di un singolo elemento d ($d =$ `sizeof`(t)), l'*indirizzo di memoria* del generico elemento `A[`i`]`, per $i = 0, \ldots, n-1$, è dato da

$$b + i * d.$$

Ad esempio, se `V` è l'array considerato nell'Esempio refes:allocazione array, l'indirizzo di `V[3]` è: $15216 + 3 * 4 = 15228$.

In C++ la memorizzazione concreta di un array non è esattamente quella sopra considerata. In C++, infatti, per un array t `A[`k`]` viene allocato, oltre allo spazio di memoria costituito dalle k celle di tipo t consecutive destinate a contenere gli elementi dell'array, anche una cella di memoria che conterrà l'indirizzo di base dell'array, cioè è un *puntatore a* t (si veda il Capitolo 7). Tutti gli accessi agli elementi dell'array avverranno in modo indiretto (ma sempre in tempo costante) attraverso la cella puntatore. La situazione è rappresentata in modo grafico nella Figura 4.1.

Nel seguito comunque potremo, in linea di massima, utilizzare gli array del C++ senza doverne necessariamente conoscere l'implementazione concreta. L'unica eccezione sarà costituita dal meccanismo di passaggio parametri (vedi par. 5.5) in cui sarà necessario prendere in considerazione le specificità dell'implementazione degli array in C++.

4.3.3 Operazioni su array

Oltre alle operazioni sui singoli elementi di un array, è, in linea di principio, possibile considerare anche operazioni che si applicano ad un'intero array (e, più in generale, ad un'intera struttura dati). Ad esempio, la stampa di un array, l'assegnamento o il confronto tra due array, ecc.

In C++, in generale, non sono previste operazioni primitive su interi array. È pertanto necessario realizzarle a programma mediante cicli che permettano di iterare su tutti gli elementi (singoli) degli array considerati. Vediamone alcuni esempi.

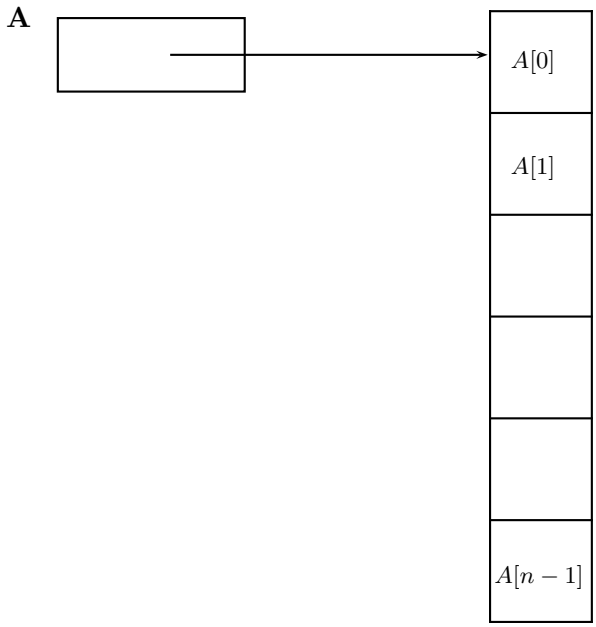


Figura 4.1: Rappresentazione grafica dell'allocazione in memoria di un array monodimensionale.

Esempio 4.8 (Assegnamento tra array) *Dati i due array*

```
int A[12] = {34,2,6,41,3,23,7,9,47,25,1,15};
int B[12];
```

l'assegnamento di (tutti gli elementi di) `A` *a* `B` *si può realizzare nel modo seguente:*

```
for (int i = 0; i < 12; i++)
  B[i] = A[i];
```

Esempio 4.9 (Confronto tra due array) *Dati i due array* `A` *e* `B` *dell'Esempio 4.8, per verificare se i due array sono uguali (cioè hanno tutti gli elementi ordinatamente uguali) e quindi stampare un opportuno messaggio, si può procedere nel modo seguente:*

```
bool array_uguali =  true;
for (int i = 0; i < 12; i++)
  if (B[i] != A[i]) {
    array_uguali = false;
    break;
  }
if (array_uguali)
```

```
    cout << ''Gli array sono uguali.''
  else
    cout << ''Gli array sono diversi.''
```

Si noti che se si scrive, ad esempio, `A = B` oppure `A == B`, con `A` e `B` array, il compilatore C++ non segnala errore, e quindi esegue le operazioni indicate ma con un significato diverso da quello che esse hanno solitamente. Di fatto, le operazioni si applicano sull'indirizzo iniziale dell'area di memoria contenente gli elementi dell'array, non sugli elementi stessi. Ad esempio, `A = B` significa che l'indirizzo iniziale di `B` diventa anche l'indirizzo iniziale di `A` e quindi che da lì in poi `A` e `B` coincidono (nel senso che condividono gli stessi elementi).

Esempio 4.10 (Input/output di array) *Il seguente codice C++ permette di leggere (da standard input) e di stampare (su standard output) tutti gli elementi dell'array* `A` *dichiarato nell'Esempio 4.8.*

```
// Operazione di input
for (int i = 0; i < 12; i++)
  cin >> A[i];

// Operazione di output (stampa)
for (int i = 0; i < 12; i++)
  cout << A[i];
```

Analogamente a quanto detto sopra per le operazioni di assegnamento e confronto, anche le operazioni di input e output predefinite (operatori `>>` e `<<`) applicate ad un intero array non funzionano correttamente. Ad esempio, l'istruzione `cout << A`, con `A` array, viene eseguita, ma il suo significato è quello di stampare l'indirizzo iniziale dell'area di memoria contenente gli elementi dell'array e non gli elementi stessi (a questo, come vedremo in un prossimo paragrafo, fanno eccezione gli array di caratteri usati per realizzare stringhe, che possono essere letti e stampati come un tutt'uno con gli operatori `>>` e `<<`).

Vediamo ora un esempio più completo di operazioni su array.

Esempio 4.11 (Minimo e massimo di una sequenza di interi)

Problema. Scrivere un programma che legga da standard input una sequenza di 10 interi e ne calcoli e visualizzi su standard output il valore minimo ed il valore massimo.

Input: una sequenza di 10 interi.
Output: i due valori interi calcolati (minimo e massimo della sequenza).
Strutture dati: un array di interi, di nome `lista`, in cui memorizzare la sequenza di interi letta da standard input.

Programma:

```
#include <iostream>
#include <climits>

using namespace std;

int main() {
  const int dim = 10;
  int lista[dim];

  for (int i = 0; i < dim; i++) {
    cout << "Inserisci l'elemento numero " << i+1 << ": ";
    cin >> lista[i];
  }

  // Stampa gli elementi dell'array 'lista'.
  cout << "L'array e' costituito dai seguenti elementi: "
       << endl;
  for (int i = 0; i < dim; i++)
    cout << lista[i] << "  ";
  cout << endl;

  // Trova il minimo.
  // La variabile 'valore_min' contiene il valore piu'
  // piccolo trovato fino a quel punto; inizializzata
  // con il massimo intero
  int valore_min = INT_MAX;
  for (int i = 0; i < dim; i++)
    if (lista[i] < valore_min)
      valore_min = lista[i];
  cout << endl << "Valore minimo: " << valore_min << endl;

  // Trova il massimo.
  // La variabile 'valore_max' contiene il valore piu'
  // piccolo trovato fino a quel punto; inizializzata
  // con il minimo intero
  int valore_max = INT_MIN;
  for (int i = 0; i < 10; i++)
    if (lista[i] > valore_max)
      valore_max = lista[i];
  cout << endl << "Valore massimo: " << valore_max << endl;

  return 0;
}
```

4.3.4 Dimensione array

La dimensione di un array (statico o semi-dinamico) deve essere specificata nel momento della sua dichiarazione e non può essere modificata successi-

vamente. Spesso però accade che la lunghezza della sequenza di dati da memorizzare nell'array non sia nota a priori. In questi casi è comunque necessario stabilire una dimensione per l'array che sia sufficiente a contenere la sequenza di lunghezza massima che si prevede possa essere fornita al programma. Il numero n degli elementi significativi presenti nell'array sarà in ogni istante minore o uguale della dimensione d dell'array stesso, lasciando indefiniti gli ultimi $d - n$ elementi dell'array.

In questi casi, la dimensione dell'array rappresenta quindi una stima per eccesso della lunghezza massima della sequenza di input e dipenderà, in generale, dalla specifica applicazione. Ad esempio, se si vuole realizzare un programma in grado di memorizzare in un array i voti in trentesimi ottenuti negli esami di un corso di laurea, per poi applicarvi diverse elaborazioni (ad esempio l'individuazione del voto minimo), allora la dimensione dell'array potrebbe essere ragionevolmente fissata a 50 che rappresenta una stima per eccesso realistica del numero massimo di voti distinti che si dovranno trattare.

Vediamo ora un esempio completo in cui la dimensione della sequenza in input non è nota a priori, ma la terminazione della sequenza stessa è determinata dalla presenza di un valore di input particolare.

Esempio 4.12 (Sequenza palindroma):

Problema. Scrivere un programma in grado di leggere una sequenza di caratteri (di lunghezza massima 100), terminata dal un carattere '.' e controllare se la sequenza è palindroma. Ad esempio, le sequenze `radar.` e `abba.` sono palindrome, mentre la sequenza `abcabc.` non lo è (n.b., il carattere `'.'` non è considerato parte della sequenza di caratteri da controllare, ma serve soltanto come delimitatore della sequenza, dato che la sua lunghezza non è specificata a priori).

Input: una sequenza di caratteri.
Output: un messaggio di risposta ("sequenza palindroma" oppure "sequenza non palindroma").
Strutture dati: un array di caratteri in cui memorizzare la sequenza di caratteri letta dallo standard input.

Programma:

```
#include <iostream>
using namespace std; int main() {
const int dim = 100;
char sequenza[dim], c;
int i = 0;
cout << "Dai sequenza di caratteri (terminata da '.') "
     << endl;
cin >> c;
while (c != '.' && i < dim)
```

```
    {sequenza[i] = c;
     i++;
     cin >> c;
    }
  // numero di caratteri componenti la sequenza letta
  int n = i;
  i = 0;
  while (i < n/2 && sequenza[i] == sequenza[n - i - 1])
      i++;
  if (i == n/2)
    cout << "sequenza palindroma" << endl;
  else
    cout << "sequenza non palindroma" << endl;
  return 0;
}
```

Data in input, ad esempio, la sequenza `radar.` il programma termina producendo il messaggio `sequenza palindroma`.

4.3.5 Array bidimensionali

Array bidimensionale. Un *array bidimensionale* $m \times n$ è una sequenza di m elementi individuati singolarmente da un numero intero i, $0 \leq i \leq m-1$, ciascuno dei quali è a sua volta una sequenza di n elementi di tipo t individuati singolarmente da un numero intero j, $0 \leq j \leq n-1$. i e j sono detti *indici* dell'array. □

In C++ un array bidimensionale di $m \times n$ elementi può essere creato tramite la seguente dichiarazione.

Dichiarazione di array bidimensionale in C++. La dichiarazione

$$t \ \texttt{B[}m\texttt{][}n\texttt{];}$$

dove:

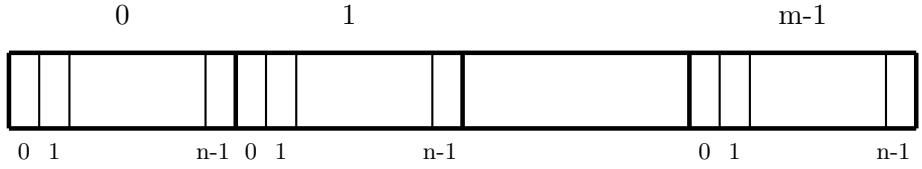
- `B` è un identificatore
- t è un tipo qualsiasi, semplice o a sua volta strutturato
- m ed n sono espressioni intere ≥ 0,

introduce un array bidimensionale di nome B, costituito da $m \times n$ elementi di tipo t. □

L'array bidimensionale `B` può essere rappresentato graficamente come una sequenza di m elementi ciascuno costituito a sua volta da una sequenza di n elementi:

Esempio 4.13 (Dichiarazione di array bidimensionale)

```
int M1[10][20];
```



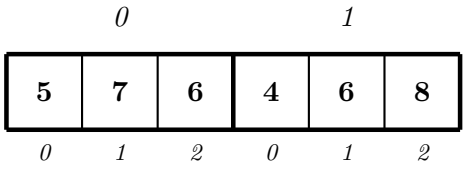
array bidimensionale di nome `M1` *costituito da* 10×20 *elementi di tipo intero (ovvero, un array di 10 elementi ciascuno di tipo array di 20 interi).*

Un array bidimensionale di $m \times n$ elementi di tipo t può anche essere inizializzato in fase di dichiarazione, fornendo una lista di m inizializzatori, ciascuno costituito da n valori di tipo t.

Esempio 4.14 (Dichiarazione di array bidimensionale inizializzato)

```
int M2[2][3] = {{5,7,6},{4,6,8}};
```

array bidimensionale di nome `M2` *costituito da* 2×3 *elementi di tipo intero inizializzati nel modo seguente:*



La selezione di un elemento di un array bidimensionale avviene con un'espressione che estende quella già vista per gli array mono-dimensionali.

Espressione con indici per array bidimensionale. Dato l'array bidimensionale

$$t \texttt{ B[}m\texttt{][}n\texttt{];}$$

con t tipo qualsiasi, l'espressione

$$\texttt{B[}e_1\texttt{][}e_2\texttt{]}$$

dove e_1 ed e_2 sono espressioni di tipo compatibile con `int`, è un'espressione con indici per `B`. Il suo significato è quello di selezionare l'elemento di indice (il valore di) e_2 all'interno dell'elemento dell'array `B` (a sua volta una sequenza) di indice (il valore di) e_1. Il tipo dell'espressione `B[`e_1`][`e_2`]` è t. □

Dunque ciascun elemento di un array bidimensionale è individuato univocamente da una coppia di numeri interi (indici) i, j.

Esempio 4.15 (Selezione di elementi di un array bidimensionale) *Dato l'array bidimensionale di interi* `M2` *mostrato nell'Esempio 4.14, le seguenti istruzioni permettono di incrementare di 1 il valore dell'ultimo elemento di* `M2` *e quindi di stamparlo sullo standard output:*

```
M2[1][2] = M2[1][2] + 1;  //ovvero, M2[1][2]++
cout << M2[1][2];         //viene stampato il valore 9
```

Altre operazioni su array bidimensionali saranno illustrate nel successivo sottocapitolo 4.4 in cui si mostrerà l'impiego degli array bidimensionali per la realizzazione della struttura dati (astratta) matrice.

Memorizzazione di un array bidimensionale

La creazione di un array bidimensionale di $m \times n$ elementi di tipo t (ovvero, t `B[`m`][`n`]`) comporta l'allocazione in memoria di m array (monodimensionali) consecutivi, ciascuno costituito da m celle di memoria consecutive di tipo t. Pertanto, supponendo che l'area di memoria in cui viene allocato `B` abbia indirizzo base b e che la dimensione (in byte) di un singolo elemento di `B` sia d ($d =$`sizeof(`t`)`), l'indirizzo di memoria del generico elemento `B[`i`][`j`]`, $i = 0, \ldots, n-1$, $j = 0, \ldots, m-1$[2], sarà dato da

$$b + (i * m + j) * d.$$

Ad esempio, per l'array `M2` precedentemente considerato, assumendo $b = 15216$, si ha che l'indirizzo di `M[1][2]` è dato da $15216 + (1 * 3 + 2) * 4$.

Array k-dimensionali

La nozione di array bidimensionale si estende facilmente a quella di array a k dimensioni con $k \geq 1$, la cui dichiarazione in C++ ha la seguente forma:

$$t \ \texttt{B[}n_1\texttt{][}n_2\texttt{]}\cdots\texttt{[}n_k\texttt{];}$$

dove n_1, n_2,...,n_k sono espressioni intere ≥ 0.

La selezione del generico elemento di un array a k-dimensioni avviene, in modo analogo a quanto mostrato per gli array bidimensionali, con una variabile con k indici, che in C++ assume la seguente forma:

$$t \ \texttt{B[}e_1\texttt{][}e_2\texttt{]}\cdots\texttt{[}e_k\texttt{]}$$

dove e_1, e_2,...,e_k sono espressioni intere qualsiasi.

4.3.6 Array "semi-dinamici"

Le attuali versioni del C++ ammettono che, sotto certe condizioni, le dimensioni di un array (mono o bi-dimensionale) possano essere specificate a run-time, ovvero, che l'espressione che rappresenta la dimensione nella dichiarazione di un array possa essere una espressione intera qualsiasi, non necessariamente costante. In questi casi parleremo di *array semi-dinamici* invece che di array statici come fatto finora.

Ad esempio, è possibile scrivere il seguente codice:

[2] N.B. Per $m = 1$ e $j = 0$ si ricade nel caso degli array ad una dimensione.

```
int n = 0;
cin >> n;
int A[n];
```

In questo caso la dimensione dell'array `A` è fornita dall'utente durante l'esecuzione del programma. L'elaborazione della dichiarazione `int A[n]` comporterà l'allocazione di un numero di elementi esattamente uguale al valore assunto dalla variabile `n`.

Riguardo l'uso, il modo in cui si opera su un array semi-dinamico è del tutto equivalente a quello previsto per un array statico. Ad esempio, l'espressione `A[i]`, con `i` variabile intera, permette di accedere all'elemento di indice `i` dell'array semi-dinamico `A` dichiarato sopra. Ovviamente, si dovrà comunque prestare attenzione, come nel caso degli array statici, a non riferirsi ad elementi non contenuti nell'array, ovvero a utilizzare valori dell'indice minori di 0 o maggiori di `n - 1`.

A differenza degli array statici, però, gli array semi-dinamici non possono essere dichiarati nello spazio delle dichiarazioni globali (vedi Capitolo 3.10). In questo caso é necessario specificare la dimensione dell'array come un'espressione costante, dato che la sua allocazione sarà necessariamente statica, ovvero effettuata a "compile-time". Ritorneremo sulla problematiche relative all'allocazione della memoria nel Capitolo 7.

Gli array semi-dinamici non possono inoltre essere utilizzati all'interno di una *struct* per specificare il tipo di uno dei campi della *struct* (si veda il Capitolo 4.6). Anche in questo caso sarà necessario utilizzare array statici.

Si presti molta attenzione al fatto che l'utilizzo di array semi-dinamici non significa che la dimensione dell'array possa essere cambiata durante l'esecuzione del programma. La dimensione dell'array deve comunque essere nota al momento della sua dichiarazione e non potrà più essere modificata durante l'esecuzione. Dunque un frammento di codice come il seguente provoca, in generale, un comportamento non corretto:

```
int n = 0;
int A[n];
cin >> n;
```

In questo caso, viene allocato un array di dimensione 0, e cioè un array che non prevede spazio di memoria per i suoi elementi. La successiva istruzione di lettura, assegna un valore ad `n`, ma questo non comporta un'allocazione di spazio per contenere gli `n` elementi dell'array `A`: `A` continua ad avere dimensione 0. Qualsiasi operazione che si riferisca ad un elemento di `A`, come, ad esempio, `A[0] = 1`, non potrà che essere logicamente errata (non verrà segnalata dal compilatore come errore, ma di fatto comportà un accesso—a "run-time"—ad una locazione di memoria non corretta).

La possibilità di utilizzare array semi-dinamici, dunque, richiede comunque la capacità di determinare la dimensione dell'array prima di creare l'array stesso. In situazioni come quella dell'esempio 4.12, in cui si deve leggere una sequenza di dati terminata da un valore speciale, la dimensione dell'array in cui memorizzare la sequenza letta non può essere determinata con precisione in anticipo. Si può soltanto, come si diceva, darne una stima per eccesso. A questo punto allora va bene utilizzare anche gli array statici.

Viceversa se l'utente e' in grado di fornire in anticipo il numero preciso di elementi che l'array dovrà contenere, allora può essere conveniente utilizzare array semi-dinamici. Si consideri ad esempio il seguente problema (generale): leggi una sequenza di n numeri interi, con n dato di input, ed effettua elaborazioni varie sulla sequenza letta (ad esempio, calcolo della media, calcolo del minimo e del massimo, ecc.). Il seguente frammento di codice C++ puo' essere utilizzato per risolvere questo problema:

```
int n;
cout << "Dai il numero di elementi da leggere";
cin >> n;
int A[n];
for (int i=0; i<n; i++)
    cin >> A[i];
// elaborazioni su A
```

In questo caso viene creato un array di esattamente n elementi. L'utente deve però essere in grado di fornire, all'inizio dell'esecuzione del programma, l'esatto numero di elementi della sequenza che verranno letti.

La possibilità di far crescere dinamicamente un array, senza dover specificare prima della sua creazione il numero degli elementi da memorizzarvi, potrà essere ottenuta soltanto con l'utilizzo delle capacità di allocazione dinamica della memoria offerte dal linguaggio, di cui tratteremo nel Capitolo 7.

4.4 Matrici

Gli array bidimensionali si prestano in modo naturale all'implementazione di una struttura dati (astratta) molto familiare in matematica ed in molti altri campi: le matrici.

Introduciamo le matrici da un punto di vista matematico, in modo rigoroso.

Matrice $m \times n$. Sia K un campo e siano m ed n due interi positivi; $m \cdot n$ scalari di K disposti nel modo seguente

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \tag{4.1}$$

costituiscono una *matrice* $m \times n$ *in* K. a_{ij} è detto *termine* o *componente* o *elemento* ij della matrice. □

Se indichiamo con A la matrice scritta nella (4.1), allora la i-esima *riga* è indicata con A_i ed è definita ponendo

$$A_i = (a_{i1}, a_{i2}, \dots, a_{in}).$$

La j-esima *colonna* è indicata da A^j ed è definita ponendo

$$A^j = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix}.$$

Vettore. Una matrice $1 \times n$ è un *vettore riga*, mentre una matrice $m \times 1$ è un *vettore colonna* (o *vettore*). □

Matrice quadrata. Una matrice $n \times n$ è detta *matrice quadrata*: in una matrice quadrata, la *diagonale principale* è formata dagli elementi a_{ij} tali che $i = j$, mentre gli elementi tali che $i + j = n + 1$ costituiscono la *diagonale secondaria*. I termini della diagonale principale sono anche detti *diagonali*. □

Esempio 4.16 (Una matrice 3×2)

$$A = \begin{pmatrix} 2 & 5 \\ 3 & 0 \\ 1 & 1 \end{pmatrix}$$

rappresenta una matrice 3×2 *di interi, i cui elementi sono:* $a_{11} = 2$, $a_{12} = 5$, $a_{21} = 3$, ...

4.4.1 Realizzazione tramite array bidimensionali in C++

Una matrice M di $m \times n$ elementi di tipo t come quella mostrata in (4.1) può essere realizzata in C++ con un array bidimensionale

t `M[`m`][`n`];`

memorizzando le m righe della matrice, ciascuna costituita da n elementi di tipo t, negli m elementi dell'array `M`, a partire dall'elemento `M[0][0]` (memorizzazione *per righe*). [3]

Esempio 4.17 *Consideriamo la matrice B di 2×3 interi*

$$B = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

In C++ questa matrice può essere realizzata con il seguente array bidimensionale

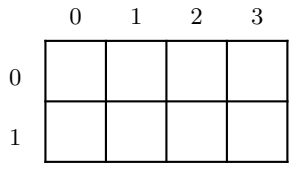
```
int B[2][3] = {{0, 1, 0}, {1, 1, 0}};
```

Si noti che questa realizzazione "diretta" della struttura dati astratta matrice tramite array bidimensionale in C++ risente delle scelte implementative della struttura dati concreta utilizzata. In particolare, risulta necessario considerare gli indici della matrice a partire da 0, piuttosto che da 1 come tipicamente avviene nella definizione astratta di matrice. Quindi, ad esempio, il primo elemento della matrice B mostrata nell'esempio 4.17 (e cioè b_{11}) sarà individuato, nella realizzazione concreta, da `B[0][0]`, mentre l'ultimo (e cioè b_{23}) sarà `B[1][2]`.

Data la corrispondenza immediata tra matrici ed array bidimensionali, così come tra vettori ed array monodimensionali, spesso si usano i termini vettore e matrice anche per riferirsi agli array. Quindi è comune dire che, ad esempio, la dichiarazione

```
int M[2][4];
```

definisce una matrice di `2` righe e `4` colonne di elementi di tipo intero e quindi rappresentare graficamente `M` come una tabella bidimensionale, organizzata in righe e colonne, analogamente a quanto visto per le matrici:



Infine si osservi che, come già sottolineato per gli array (si veda il paragrafo 4.3.4), nel caso in cui la dimensione esatta della matrice non sia nota a priori, si può comunque realizzare la matrice tramite un array bidimensionale di dimensioni ritenute sufficienti (stima per eccesso), utilizzando

[3] In alternativa, ma meno comunemente, la matrice M potrebbe essere memorizzata *per colonne* in un array bidimensionale t `M[`n`][`m`]`.

eventualmente solo una parte dell'array per contenere gli elementi della matrice e memorizzando il numero effettivo dei suoi elementi (che dovrà essere sicuramente minore o uguale alle dimensioni dell'array utilizzato) in due variabili intere che completano l'implementazione della matrice.

4.4.2 Operazioni su matrici

Mostriamo ora l'implementazione in C++ di alcune tipiche operazioni su matrici, assumendo che quest'ultime siano realizzate tramite array bidimensionali nel modo precedentemente illustrato.

Esempio 4.18 (Lettura e scrittura di una matrice)

Il seguente programma permette di leggere da standard input una sequenza di numeri interi, di memorizzarli in una matrice di nome `mat`*, e quindi di stampare su standard output la matrice* `mat`*, una riga della matrice per ogni linea di stampa. Il numero di righe e di colonne della matrice non è noto a priori e viene richiesto all'utente all'inizio del programma. È però stabilito un limite massimo alle dimensioni della matrice (fissato ad esempio in* 100×100*).*
Problema*. Scrivere un programma che legga, memorizzi e stampi una matrice di interi.*

Input*: numero di righe e di colonne della matrice; elementi della matrice.*
Output*: matrice in forma tabellare.*
Strutture dati*: array bidimensionale.*

Programma*:*

```
#include <iostream>
using namespace std;

const int MAX_RIGHE = 100
const int MAX_COLONNE = 100

int main() {

  // Richiesta delle dimensioni della matrice
  int num_righe;
  do {
    cout << "Numero di righe (max " << MAX_RIGHE << "): ";
    cin >> num_righe;
  } while (num_righe < 1 || num_righe > MAX_RIGHE);
  int num_colonne;
  do {
    cout<<"Numero di colonne (max "<<MAX_COLONNE << "): ";
    cin >> num_colonne;
  } while (num_colonne < 1 || num_colonne > MAX_COLONNE);
```

```
  // Dichiarazione e caricamento della matrice
  int mat[MAX_RIGHE][MAX_COLONNE];
  for (int i = 0; i < num_righe; i++)
    for (int j = 0; j < num_colonne; j++) {
      cout << "Inserisci l'elemento (" << i << ",
                                      " << j << "): ";
      cin >> mat[i][j];
    }

  // Stampa
  for (int i = 0; i < num_righe; i++) {
    cout << endl;
    for (int j = 0; j < num_colonne; j++) {
      cout.width(7);
      cout << mat[i][j];
    }
  }
  cout << endl;

  // Eventuali elaborazioni sulla matrice

  return 0;
}
```

I due esempi seguenti mostrano altrettante possibili operazioni su matrici. I frammenti di codice mostrati possono facilmente essere inseriti nel programma dell'Esempio 4.18 nella parte in cui sono previste "eventuali elaborazioni sulla matrice".

Esempio 4.19 (Trasposta di una matrice)
Problema. *Scrivere un programma che calcoli la trasposta di una matrice fornita dall'utente.*

Programma:

```
  ...
  // Dichiarazione e calcolo della matrice trasposta
  int mat_trasposta[MAX_COLONNE][MAX_RIGHE];
  for (int i = 0; i < num_righe; i++)
    for (int j = 0; j < num_colonne; j++)
      mat_trasposta[j][i] = mat[i][j];

  // Stampa la matrice trasposta.
  cout << endl << "\t *** MATRICE TRASPOSTA ***" << endl;
  for (int i = 0; i < num_colonne; i++) {
    cout << endl;
    for (int j = 0; j < num_righe; j++)
      cout << setw(6) << mat_trasposta[i][j];
  }
```

```
cout << endl << endl;
...
```

Esempio 4.20 (Sommatoria diagonale di una matrice quadrata)
Problema*. Scrivere un programma che calcoli e stampi la somma degli elementi sulla diagonale principale di una matrice quadrata fornita dall'utente.*

Programma*:*

```
...
// Calcolo della somma degli elementi sulla diagonale.
int somma = 0;
for (int i = 0; i < num_righe_colonne; i++)
  somma += mat[i][i];

cout<<"Somma degli elementi sulla diagonale principale: "
    << somma << "." << endl;
...
```

Altre tipiche operazioni del calcolo matriciale, come ad esempio somma, differenza, prodotto tra matrici, inversione, dipendenza lineare, ecc., possono essere realizzate come programmi/funzioni C++ in modo del tutto analogo a quanto visto sopra e sono pertanto lasciate come (utile) esercizio al lettore.

4.5 Stringhe

Un'altra struttura dati di uso molto comune, facilmente realizzabile tramite array, è quella delle stringhe.

Una *stringa* è una sequenza di n caratteri adiacenti, con $n \geq 0$.

Ad esempio, `"ciao"` è una stringa di 4 caratteri, mentre `""` è una stringa di 0 caratteri (detta *stringa vuota*).[4]

Tipiche operazioni su stringhe, oltre a quelle generali di assegnamento, di confronto e di input/output, sono le operazioni di calcolo della lunghezza di un stringa, di concatenazione tra due stringhe, di ricerca di una sottostringa in una stringa data. E' inoltre comune dover accedere ad un singolo carattere di una stringa, specialmente al suo carattere di testa, o dover aggiungere o rimuovere un singolo carattere da una stringa data.

[4] I doppi apici sono usati, nella tipica notazione per le stringhe, come delimitatori della stringa e pertanto non fanno parte di essa.

4.5.1 Realizzazione tramite array

Una stringa può essere facilmente realizzata tramite un array di caratteri. Questa è ad esempio l'implementazione delle stringhe standard utilizzata in C ed è quella che esamineremo in questo capitolo. Altre implementazioni sono ovviamente possibili. In C++, in particolare, la stringa è vista come un tipo di dato astratto e quindi realizzata tramite un'opportuna classe (denominata `string`) che di fatto nasconde all'utente l'implementazione concreta della struttura dati (astratta) stringa. Ritorneremo sulla realizzazione delle stringhe del C++ in un capitolo successivo. Per ora esaminiamo in dettaglio l'implementazione fornita dal C. Si noti che essendo il C++ un sovrainsieme del C è comunque sempre possibile utilizzare questa implementazione anche all'interno dell'ambiente C++.

Una stringa "$c_1 c_2 \ldots c_n$" di n caratteri è realizzata in C tramite un array di k elementi di tipo `char`, con $k > n$, costituiti nell'ordine dagli n caratteri della stringa, seguiti dal carattere speciale `'\0'` per indicare la fine della stringa.

0	1	$\cdots$	$n-1$	n	$n+1$	$\cdots$	k
'c_1'	'c_2'	$\cdots$	'c_n'	'\0'	-	$\cdots$	-

Una stringa di nome `s` può pertanto essere dichiarata in C (C++) come un normale array di caratteri, con un'opportuna dimensione (maggiore della lunghezza massima prevista per la stringa), opportunamente inizializzata con i caratteri componenti la stringa stessa più il delimitatore `'\0'`.

Esempio 4.21 *La dichiarazione*

```
char s[10] = {'c','i','a','o','\0'};
```

definisce una stringa di nome `s` *con valore (iniziale)* `"ciao"` *e una lunghezza massima di 9 caratteri (si noti che, come già osservato per gli array in generale, anche in questo caso è possibile fornire una lista di inizializzatori con un numero di elementi inferiore alla dimensione dell'array):*

s

0	1	2	3	4	5	6	7	8	9
'c'	'i'	'a'	'o'	'\0'	-	-	-	-	-

Il C++, così come la maggior parte dei linguaggi di programmazione, prevede anche la possibilità di utilizzare apposite *costanti stringa*, in cui la stringa viene scritta con la più naturale notazione tra doppi apici. Le

costanti stringa possono essere usate sia come inizializzatori in una dichiarazione di una variabile, sia all'interno di un'espressione che preveda l'uso di questo tipo di dato. Ad esempio, la dichiarazione dell'Esempio 4.21, può essere scritta, equivalentemente, come

```
char s[10] = "ciao";
```

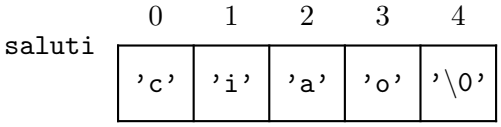
In questo caso il delimitatore `'\0'` verrà inserito automaticamente al termine della stringa (la rappresentazione interna della stringa e', comunque, la stessa mostrata nell'Esempio 4.21).

Nota. Una costante di stringa può contenere qualsiasi carattere, compresi lo spazio ed i caratteri speciali come `'\n'` ("a capo"), `'\t'` ("tabulazione"), ma non `'\0'`.

È possibile dichiarare una stringa anche senza specificare esplicitamente la dimensione dell'array che la contiene; in questo caso, come per un qualsiasi array con inizializzazione esplicita, la dimensione sarà quella dedotta dal numero di elementi presenti nell'inizializzatore (nel caso delle stringhe bisogna ricordarsi sempre che c'è anche da contare il delimitatore di stringa `'\0'`). Ad esempio, con la dichiarazione

```
char saluti[] = "ciao";
```

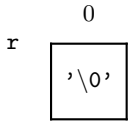
vengono allocati esattamente 5 byte nel modo seguente:



Nota. La costante di stringa `""` indica la *stringa vuota* e può essere usata come qualsiasi altra costante di stringa. Ad esempio la dichiarazione

```
char r[] = "";
```

definisce una stringa di nome `r`, inizializzata a stringa vuota. La sua rappresentazione interna è pertanto la seguente:



Nota. Attenzione a non confondere le *costanti carattere* con le *costanti stringa*. Ad esempio, la costante `'a'` e la costante `"a"` denotano oggetti profondamente diversi: la prima, `'a'`, è un carattere singolo, rappresentato dal suo codice ASCII (un intero tra 0 e 255), memorizzato in un singolo byte; il secondo, `"a"`, è un stringa di lunghezza 1 e come tale memorizzata in un array di due elementi di cui il primo è il carattere `'a'` ed il secondo è il delimitatore di stringa `'\0'`.

4.5.2 Operazioni su stringhe

Le stringhe stile C che abbiamo presentato finora sono a tutti gli effetti array di caratteri e come tali non prevedono alcuna operazione primitiva (e cioè, direttamente supportata dal linguaggio), a parte l'accesso al singolo elemento tramite l'operazione di selezione prevista dagli array. Il C++ mette però a disposizione alcune funzioni predefinite, facenti parte di librerie standard, con cui poter operare più facilmente su stringhe stile C. Precisamente, sono disponibili funzioni per:

- input/output di stringhe, nella libreria `iostream`;
- altre operazioni tipiche della manipolazione di stringhe, nella libreria `cstring`.

Scrittura di una stringa tramite <<

L'operatore di output `<<` che già conosciamo si applica in modo naturale anche alle stringhe stile C. Vediamo un esempio.

L'esecuzione delle istruzioni:

```
char msg[100]="errore";
cout << msg << endl;
```

causa l'invio sul monitor (lo standard output di default) del messaggio:

```
errore
```

seguito da un carattere di "a capo".

Lo stesso effetto si può ottenere anche scrivendo direttamente la costante di stringa `"errore"` come espressione dell'operatore `<<`:

```
cout << "errore" << endl;
```

oppure, inserendo il carattere speciale che provoca l'"a capo" direttamente nella stringa da stampare:

```
cout << "errore\n";
```

Lettura di una stringa tramite >>

Anche l'operatore di input `>>` già utilizzato per i tipi semplici primitivi si applica alle stringhe stile C. Vediamo un esempio.

L'esecuzione delle istruzioni:

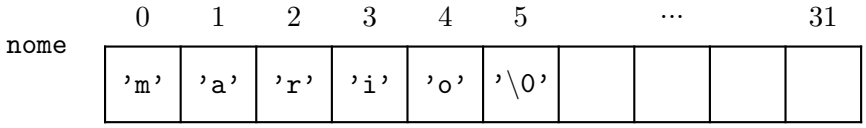
```
char nome[32];
cin >> nome;
```

provoca la lettura, uno dopo l'altro, dei caratteri presenti sullo standard input fino ad incontrare il carattere "spazio" o "a capo" e li assegna nell'ordine all'array `nome`.[5] Al termine, viene inserito automaticamente il carattere di fine stringa `'\0'`.

Supponiamo che l'utente digiti la sequenza di caratteri

```
mario bianchi
```

Al termine della lettura la stringa `nome` avrà il seguente contenuto:



I caratteri rimasti sullo stream di input potranno essere letti con successive operazioni di input. Se invece venisse digitata, ad esempio, la sequenza `mario_bianchi` (seguita, al solito, dal comando di "a capo"), allora tutta la sequenza verrebbe memorizzata come stringa nell'array `nome`, con il carattere `'\0'` inserito, automaticamente, al termine della sequenza.

Nota. Si osservi che è necessario, in generale, dichiarare l'array utilizzato per memorizzare la stringa di dimensioni adeguate a contenere la stringa stessa. Dichiarare l'array, ad esempio, senza dimensioni e quindi memorizzarvi una stringa come nella sequenza di istruzioni

```
char nome[];
cin >> nome;
```

è sicuramente un errore e può portare ad un comportamento anomalo del programma, dato che l'operazione di lettura preleva comunque i caratteri presenti sullo stream di input e li memorizza nell'area di memoria individuata da `nome`: siccome l'indirizzo memorizzato in `nome`, e quindi l'area in cui verranno scritti i caratteri della stringa, è indefinito, il risultato del programma è del tutto imprevedibile. Problemi analoghi si presentano anche quando il numero di caratteri che vengono letti dallo stream di input eccede la dimensione massima prevista per l'array.

Vediamo un semplice esempio in cui si richiede di leggere e scrivere una stringa.

Esempio 4.22 *(Iniziale maiuscola)*

```
/* Leggi una stringa (max. 50 caratteri) e stampala con
   iniziale maiuscola */

#include <iostream>
using namespace std;
```

[5] La lettura termina anche se si incontra il carattere di "end_of_file", come vedremo meglio in un successivo capitolo dedicato all'input/output da file.

```
int main() {
  const int dim = 51;
  char s[dim];
  cout << "Inserire stringa (max " << dim-1 << " caratteri): ";
  cin >> s;
  if (s[0]>='a' && s[0]<='z')
     s[0] = s[0] - 32;
  cout << s << endl;
  return 0;
}
```

Se l'input fornito al programma è, ad esempio, la stringa

```
carlo
```

l'output prodotto sarà

```
Carlo
```

Nell'esempio di sopra si nota tra l'altro che è possibile accedere al singolo elemento della stringa (ad esempio il suo carattere iniziale) tramite la normale operazione di selezione tramite indice prevista per gli array.

L'utilizzo dell'operatore `>>` per leggere stringhe pone alcuni problemi, e precisamente.

- La lettura termina appena si incontra sullo stream di input un carattere "spazio" o "a capo". Ad esempio, con riferimento all'Esempio 4.22, se l'input fornito fosse

  ```
  frase di prova
  ```

  l'output prodotto sarebbe

  ```
  Frase
  ```

  Inoltre solo i caratteri "spazio" o "a capo" possono fungere da "delimitatori" della stringa in input.

- Se la lunghezza della stringa in input è superiore alla dimensione dell'array in cui la stringa viene memorizzata la lettura dei caratteri in eccesso viene effettuata lo stesso, compromettendo la correttezza del programma (vedi nota precedente). Per evitare che si verifichi questa situazione bisogna prevedere la lunghezza massima della stringa e dimensionare l'array di conseguenza (e cioè dare una dimensione uguale o maggiore della massima lunghezza della stringa). In ogni caso non c'è modo di interrompere la lettura se il numero dei caratteri letti eccede quello previsto.

Per ovviare a questi problemi si può utilizzare un'altra funzione predefinita offerta dalla libreria `iostram`: la funzione `getline`.

Lettura di una stringa tramite `getline`

La funzione `getline` ha la seguente forma generale:

```
s.getline(str,l,d)
```

dove:

- `s` è uno stream di input (ad esempio, `cin`),
- `str` è una stringa tipo C,
- `l` è il numero di caratteri da estrarre dallo stream di input `s`,
- `d` (opzionale) è il carattere delimitatore della stringa in input; se assente, si assume come suo valore il carattere `'\n'` ("a capo").

L'esecuzione di questa funzione comporta le seguenti azioni: i caratteri sullo stream di input `s` vengono estratti, uno alla volta, ed assegnati ordinatamente a `str`, finchè non si verifica una delle seguenti condizioni:

- il carattere estratto coincide con il carattere delimitatore `d` (in questo caso il carattere estratto non viene aggiunto alla stringa memorizzata in `str`)
- il numero di caratteri estratti è pari a `l` -1
- è stato raggiunto l'end_of_file sullo stream di input.

Al termine della lettura viene aggiunto alla stringa memorizzata in `str` il carattere speciale `'\0'` a indicare la fine della stringa. Si osservi che, come per gli altri operatori e funzioni di lettura dati, se sullo stream di input il dato atteso non è (ancora) presente, anche nel caso della `getline` l'esecuzione si sospende in attesa del dato da leggere.

È evidente che l'utilizzo della `getline` risolve entrambi i problemi evidenziati sopra per l'operatore `>>`. Precisamente, la lettura tramite `getline` non si arresta al primo carattere "spazio" o "a capo" incontrato sullo stream di input, ma soltanto quando si incontra il carattere delimitatore specificato nella chiamata della `getline`. Inoltre, se l'array `sttr` ha dimensione `l`, allora non si corre il rischio di andare a memorizzare i caratteri letti al di fuori dell'area di memoria ad essi riservata. A questo proposito si noti che la condizione prevista nella `getline` sul numero dei caratteri estratti (uguale a `l` -1) tiene conto del fatto che ai caratteri veri e propri della stringa andrà aggiunto il carattere delimitatore `'\0'`.

Come esempio di uso della `getline` consideriamo nuovamente il problema dell'Esempio 4.22 e mostriamo come risolverlo utilizzando `getline` invece che l'operatore `>>`. L'unica modifica riguarda l'istruzione relativa alla lettura della stringa `s` che in questo caso diventa:

```
cin.getline(s,dim,'\n');
```

Se l'input fornito al programma fosse, ad esempio,

```
frase di prova
```

l'output prodotto sarebbe

```
Frase di prova
```

Da notare che si può facilmente specificare un diverso carattere come terminatore della stringa di input semplicemente specificando il carattere voluto come terzo parametro della `getline`.[6] Ad esempio, nel programma di sopra potremmo indicare che la stringa da leggere è terminata dal carattere `'.'` (punto) semplicemente specificando la `getline` nel modo seguente: `cin.getline(s,dim,'.')`. In questo caso, se l'input fornito fosse

```
frase di
prova.
```

l'output prodotto sarebbe

```
Frase di
prova
```

Nota. Si presti attenzione al fatto che, sebbene la `getline` effettui un controllo sul numero dei caratteri letti e quindi memorizzati nell'array destinato a contenere la stringa, il corretto dimensionamento dell'array stesso è comunque a carico del programmatore. Ad esempio, il seguente codice:

```
char s[];
 ...
cin.getline(s,51,'\n');
```

è chiaramente una situazione d'errore dato che la dichiarazione dell'array non alloca memoria per l'array stesso mentre la `getline` prevede che vengano letti e memorizzati 50 caratteri nell'area di memoria denotata da `s`.

Lettura carattere a carattere. Essendo le stringhe stile C array di caratteri è anche possibile operare su esse come normali array. In particolare, l'operazione di input di una stringa può essere realizzata con un ciclo di operazioni `get`, una per ogni singolo carattere della stringa da leggere, come mostrato nella seguente nuova versione dell'Esempio 4.22.

```
int main() {
  const int dim = 51;
  char s[dim];
  cout << "Inserire stringa (max " << dim-1 << " caratteri): ";
  char c;
  c = cin.get();
  while (c != '\n' && i < 50) {
     s[i] = c;
     i++;
     c = cin.get();
  }
  s[i] = '\0';
  if (s[0]>='a' && s[0]<='z')
     s[0] = s[0] - 32;
```

[6] Si osservi che con la `getline` è comunque possibile specificare un solo carattere delimitatore alla volta, non un insieme di possibili caratteri delimitatori.

```
    cout << s << endl;
    return 0;
}
```

Si noti l'inserimento "a mano" del delimitatore di stringa `'\0'` al termine della lettura della stringa.

Altre operazioni su stringhe

Molte delle tipiche operazioni previste per la struttura dati stringa sono fornite come funzioni predefinite della libreria `cstring`. Il loro utilizzo richiede pertanto la presenza della direttiva di preprocessing:

```
#include <cstring>
```

Esaminiamo sinteticamente alcune di queste funzioni (quelle che utilizzeremo negli esempi in questo testo), rimandando a fonti più specialistiche per un trattamento più completo e approfondito di queste e delle altre funzioni della libreria `cstring`.

Nel seguito `s` e `r` verranno utilizzati per indicare due stringhe tipo C.

- `strlen(s)`: restituisce come suo risultato la lunghezza della stringa `s`. Ad esempio:

  ```
  char s[10] = "ciao";
  cout << strlen(s);
  ```

  stampa 4.

- `strcpy(s,r)`: copia la stringa `r` nella stringa `s`. Ad esempio:

  ```
  char s[20];
  char r[] = "una frase";
  strcpy(s,r);
  cout << s;
  ```

  stampa la stringa `una frase`.

- `strcat(s,r)`: concatena la stringa `r` alla stringa `s` in `s`. Ad esempio:

  ```
  char s[32] = "ciao";
  char r[] = " mondo!";
  strcat(s,r);
  cout << s;
  ```

  stampa la stringa `ciao mondo!`. Si noti che l'array in cui viene memorizzata la stringa risultante dalla concatenazione deve essere dimensionato adeguatamente. Precisamente la sua dimensione deve essere (strettamente) maggiore della somma delle dimensioni delle due stringhe che vengono concatenate.

- `strcmp(s,r)`: confronta le due stringhe `s` ed `r` e restituisce:

- un risultato $= 0$ se `s` ed `r` sono identiche
- un risultato < 0 se `s` precede `r`
- un risultato > 0 se `r` precede `s`.

dove l'ordinamento tra stringhe utilizzato è quello del normale ordinamento lessicografico tra stringhe, secondo cui, ad esempio, la stringa `casa` precede la stringa `cavallo`, così come la stringa `casato`, ma non la stringa `dado`.

Ad esempio:

```
char s[32] = "test";
char r[100];
cin >> r;
if (strcmp(s,r) == 0)
cout << "stringhe uguali" << endl;
```

Se la stringa data in input è la stringa `test` il programma stamperà il messaggio `stringhe uguali`.

Mostriamo un semplice programma completo che prevede l'utilizzo di funzioni della libreria `cstring` (in particolare, della funzione `strcmp`).

Esempio 4.23 *(Controllo password) Leggi una password (stringa di max. 8 caratteri) fornita dall'utente e confrontala con la password memorizzata nel programma. Se la password letta è diversa da quella memorizzata, riprova (massimo 3 tentativi).*

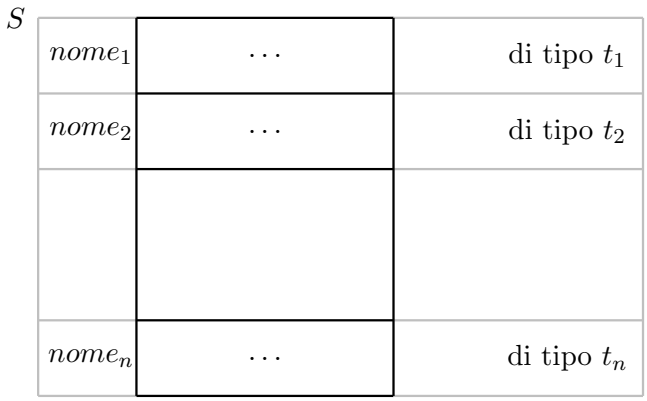
```
int main() {
  char password[] = "nSdW28ht";
  char passwd_utente[9];
  cout << "Inserire password (max 8 caratteri): ";
  int tentativi = 0;
  do {
     cin >> passwd_utente;
     if (strcmp(passwd_utente, password) == 0);
        break;
     else {
        tentativi++;
        if (tentativi < 3)
           cout << "Password non valida, ritenta" << endl;
     }
  } while (tentativi < 3);
  if (tentativi == 3)
     cout << "Accesso negato" << endl;
  else
     cout << "Accesso consentito" << endl;
  return 0;
}
```

4.6 *struct*

Un altro importante costruttore di tipo che permette di definire tipi strutturati in C++ è il costruttore `struct`. Con questo costruttore è possibile creare strutture dati (concrete), che indicheremo semplicemente con il nome *struct*, definite nel modo seguente.

struct. Un oggetto di tipo *struct* (o, più semplicemente, una *struct*) è una sequenza di n elementi ($n \geq 0$), rispettivamente di tipo $t_1, \ldots, t_n$ (non necessariamente distinti), individuati singolarmente da n nomi simbolici $nome_1, \ldots, nome_n$. □

Una tipica rappresentazione grafica per una *struct* con n elementi è mostrata nella figura seguente.



dove S è il nome della struttura dati e $nome_1, \ldots, nome_n$ sono gli elementi della struttura dati (detti anche *campi* della struttura).

4.6.1 Dichiarazione di *struct* in C++

Una variabile di tipo (definito tramite il costruttore di tipo) `struct` può essere creata tramite la seguente dichiarazione.

Dichiarazione di variabile di tipo *struct*. La dichiarazione

```
struct { t1 nome1;
         t2 nome2;
         ...
         tn nomen;} S;
```

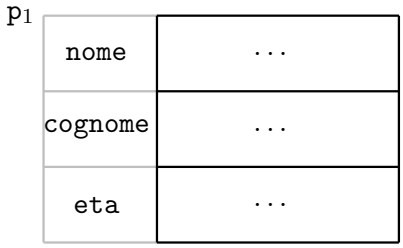
dove: `S`: identificatore di variabile
`t1`,...,`tn`: tipi qualsiasi (primitivi o definiti da utente)
`nome1`,...,`nomen`: identificatori

crea una variabile di nome S, di tipo *struct* costituito da n elementi di nome $\texttt{nome}_1,\ldots,\texttt{nome}_n$, rispettivamente di tipo $\texttt{t}_1,\ldots,\texttt{t}_n$. □

Esempio 4.24 *(Dichiarazione di variabile di tipo struct)*

```
struct {char nome[32];
        char cognome[32];
        int eta;} p1;
```

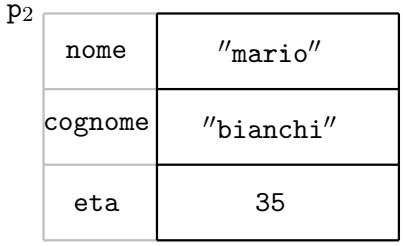
crea una variabile di nome $\texttt{p}_1$ *e tipo* `struct {...}` *(*$\texttt{p}_1$ *= nome della variabile;* `struct {...}` *= tipo della variabile). Rappresenta una struttura dati struct di nome* $\texttt{p}_1$ *costituita da due campi di tipo array di caratteri e un campo di tipo intero, rispettivamente con nomi* `nome`*,* `cognome` *ed* `eta`*.*



Come per gli array, anche in una dichiarazione di *struct* è possibile specificare dei valori di inizializzazione.

Esempio 4.25 *(Dichiarazione di struct con inizializzazione)*

```
struct {char nome[32];
        char cognome[32];
        int eta;} p2 = {"mario","bianchi",35};
```



Come fatto finora, la dichiarazione di una variabile di tipo *struct* introduce sia il nuovo tipo, sia la variabile di quel tipo (come già visto per array).

È possibile (preferibile) separare i due momenti: prima si definisce il nuovo tipo e gli si associa un nome; poi si dichiarano una o più variabili di quel tipo.

Dichiarazione di tipo *struct*. La dichiarazione

```
struct NomeTipo {t1 nome1;
                 t2 nome2;
                 ...
                 tn nomen;};
```

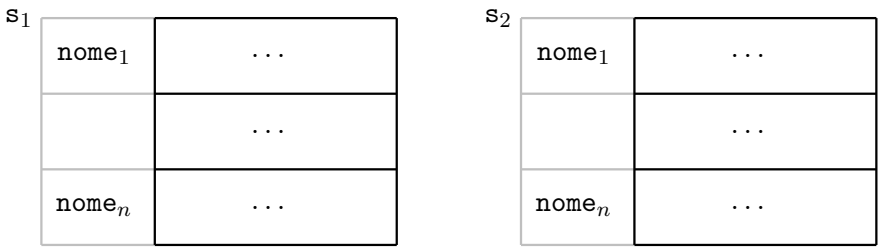
dove `NomeTipo` è un identificatore qualsiasi, definisce un nuovo tipo *struct*, composto dai campi $\texttt{nome}_1$,...,$\texttt{nome}_n$, rispettivamente di tipo $\texttt{t}_1$,...,$\texttt{t}_n$, e gli associa il nome `NomeTipo` (di fatto `NomeTipo` è un *sinonimo* del tipo costruito tramite `struct`). □

Si noti che questa dichiarazione non alloca memoria; definisce soltanto un nuovo tipo e gli associa un nome.

A questo punto è possibile dichiarare una o più variabili del tipo introdotto in precedenza:

```
NomeTipo s1;     //struct di nome s1 e tipo NomeTipo
NomeTipo s2;     //struct di nome s2 e tipo NomeTipo
```

Si noti che questa dichiarazione, invece, alloca memoria, come mostrato graficamente qui di seguito.



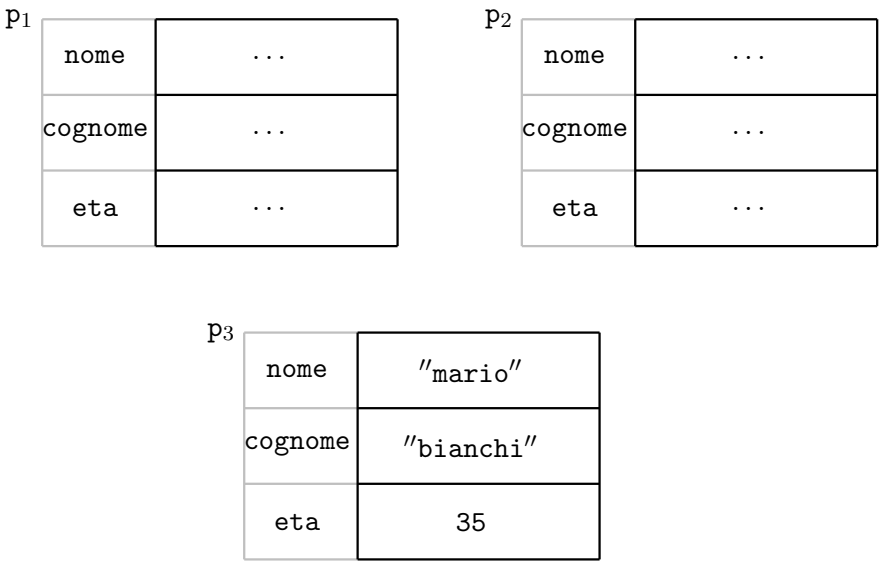
Esempio 4.26 *(Dichiarazione di tipo struct* `persona`*)*

```
struct persona {char nome[32];
                char cognome[32];
                int eta;};
```

→ *nuovo tipo di nome* `persona`

```
persona p1 ;
persona p2 ;
persona p3 = {"mario","bianchi",35};
```

→ *tre variabili di tipo struct* `persona`



Si noti che p_1, p_2 *e* p_3 *hanno tutti e tre lo stesso tipo* `persona`.

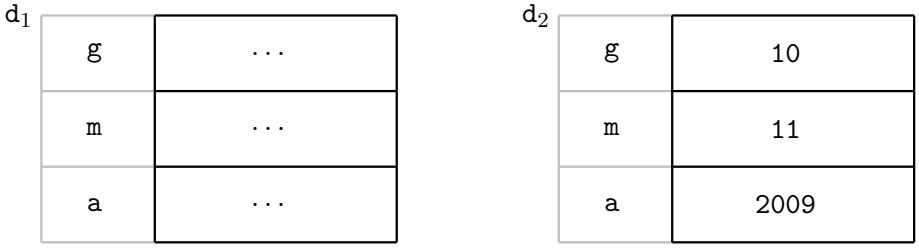
Esempio 4.27 *(Dichiarazione di tipo struct* `data` *= data del giorno)*

```
struct data {int g;
             int m;
             int a;};
```

→ *nuovo tipo di nome* `data`

```
data d1 ;
data d2 = {10, 11, 2009};
```

→ *due variabili di tipo struct* `data`.



La dichiarazione di un tipo *struct* deve apparire, ovviamente, prima del suo uso. In particolare, può essere posta all'interno del corpo di una funzione (anche del `main`), se si vuole che sia visibile soltanto in essa, oppure, come accade più spesso, tra le dichiarazioni globali, se si vuole che sia visibile in tutto il programma. Ad esempio:

```
#include <iostream>
using namespace std;
struct data {int g;
             int m;
             int a;};
int main() {
    data d1,d2;
    ...
```

Infine si noti che il tipo dei campi di una *struct* è un tipo qualsiasi e quindi in particolare può essere un tipo strutturato, quale un array o un'altra *struct*.

Un esempio di *struct* contenente un array è quello della *struct* `persona` mostrato nell'Esempio 4.26: i campi `nome` e `cognome` sono array di caratteri.

È possibile utilizzare anche *struct* all'interno di altre *struct*. Ad esempio, nella dichiarazione:

```
struct dati_anagrafici {
                char nome[32];
                char cognome[32];
                int eta;
                char sesso;
                data data_nascita;
               };
```

si utilizza il tipo *struct* `data` come tipo di uno dei campi della *struct* `dati_anagrafici`. Le due dichiarazioni seguenti introducono due variabili di tipo `dati_anagrafici`:

```
dati_anagrafici A;
dati_anagrafici B = {"mario","bianchi", 32,'m',{30,8,1975}};
```

4.6.2 Operazioni su *struct*

Operazione di selezione

Operazione fondamentale per poter utilizzare una *struct* è, come nel caso degli array, quella di *selezione* di un suo singolo elemento. Nel caso delle

struct l'operazione di selezione avviene semplicemente tramite il nome dell'elemento stesso. Si tratta di un'operazione con *accesso diretto*, il cui tempo d'esecuzione non dipende dalla posizione relativa dell'elemento all'interno della *struct*.

In C++ la selezione di un elemento di una *struct* avviene tramite un'opportuna espressione.

Espressione con campo. Data la *struct*

```
struct NomeTipo {t_1  nome_1;
                 t_2  nome_2;
                 ...
                 t_n  nome_n;};
NomeTipo s;
```

l'espressione:

```
s.nome_i
```

seleziona il campo `nome`$_i$ della struttura dati *struct* `s`. Il tipo dell'espressione `s.nome`$_i$ è t_i. □

Si presti attenzione al fatto che la selezione si applica a *variabili* di tipo *struct*, non a *tipi struct*. Dunque un'espressione come, ad esempio, `NomeTipo.c`$_i$ è priva di significato.

Esempio 4.28 *(Selezione di elementi in una struct) Siano* `A` *e* `B` *le due struct di tipo* `dati_anagrafici` *mostrate sopra. Le seguenti istruzioni contengono espressioni di accesso (sia in lettura che in scrittura) ai campi di queste due struct:*

```
A.eta = 21;
A.eta = A.eta + 1; (o, in modo equiv.: A.eta++)
cout << "Iniziale cognome: " << B.cognome[0];
cout << "Anno nascita: " << B.data_nascita.a;
```

Nel terzo esempio si accede al campo `cognome` di `B` che è a sua volta una struttura dati, precisamente un array, e di questo si considera l'elemento di indice 0. Anche nell'ultimo esempio si accede ad un campo che è a sua volta una struttura dati, precisamente una *struct* di tipo `data`, e di questa si considera l'elemento di nome `anno`. Si noti che l'operatore '.' è associativo a sinistra.

Altre operazioni primitive su *struct*

Mentre gli array non ammettono alcuna operazione primitiva che permetta di operare sull'intera struttura dati, nel caso delle *struct* è possibile eseguire l'assegnamento tra (intere) *struct*, purchè dello stesso tipo.

Ad esempio, se `p`$_2$ e `p`$_3$ sono le due *struct* di tipo `persona` definite in precedenza, è possibile scrivere:

```
p2 = p3;
```

L'assegnamento copia campo a campo i valori della *struct* di destra nella *struct* di sinistra.

Si noti che l'assegnamento avviene anche se alcuni campi delle *struct* coinvolte sono array (come nel caso di `persona`) (si ricordi che, in generale, l'assegnamento tra array non è previsto come operazione primitiva).

A parte l'assegnamento, non è prevista nessun'altra operazione, nè primitiva nè di libreria standard, su intere *struct*. In particolare non sono previste operazioni di input/output, nè operazioni di confronto. Tutte queste operazioni devono essere realizzate a programma, operando campo a campo, come mostrato negli esempi che seguono.

4.6.3 Esempi

Esempio 4.29 *(Leggi e stampa i dati di una persona)*

```
/* Leggi da std input i dati di una persona e stampali su std output
   con opportuno formato */

#include <iostream>

using namespace std;

struct persona {
     char nome[32];
     char cognome[32];
     int eta;
  };

 int main() {

    persona p;

    // lettura dati persona
    cout << "Dai il nome: ";
    cin >> p.nome;
    cout << "Dai il cognome: ";
    cin >> p.cognome;
    cout << "Dai l'eta': ";
    do cin >> p.eta; while (p.eta < 0 || p.eta > 150);

    // stampa dei dati letti
    cout << "\nNOME: " << p.nome << endl;
    cout << "COGNOME: " << p.cognome << endl;
    cout << "ETA': " << p.eta << endl;
```

```
   return 0;
}
```

Esempio 4.30 *(Data maggiore)*

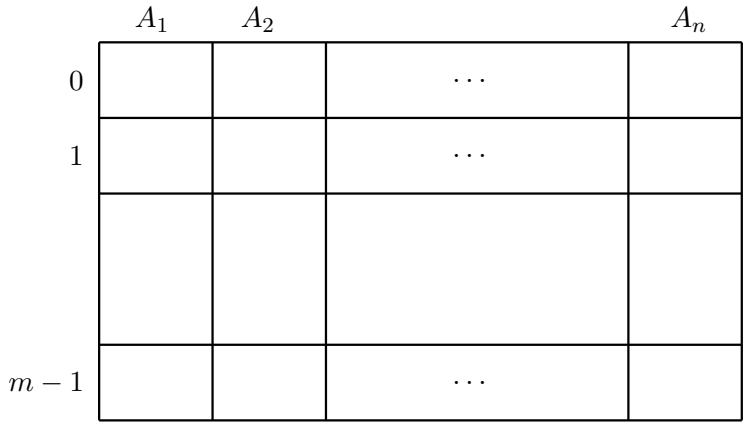
```
/* Leggi due date (G M A) e determina se le due date sono uguali o qual'e'
  la maggiore delle due. Ad es.: 1 8 1975 > 12 10 1972. */

#include <iostream>
using namespace std;

struct data {
     int g;
     int m;
     int a;
  };

int main() {
  data d1, d2;

  // lettura date
  cout << "Dai la prima data (g m a): ";
  cin >> d1.g >> d1.m >> d1.a;
  cout << "Dai la seconda data (g m a): ";
  cin >> d2.g >> d2.m >> d2.a;

  // confronto date lette
  if (d1.a == d2.a && d1.m == d2.m && d1.g == d2.g)
     cout << "Le due date sono uguali" << endl;
  else {
     if (d1.a > d2.a ||
         d1.a == d2.a && d1.m > d2.m ||
         d1.a == d2.a && d1.m == d2.m && d1.g > d2.g)
         cout << "La data maggiore e' "
              << d1.g << '/' << d1.m << '/' << d1.a << endl;
     else
         cout << "La data maggiore e' "
              << d2.g << '/' << d2.m << '/' << d2.a << endl;
  }

  return 0;
}
```

4.7 Tabelle

La struttura dati *struct*, combinata con *array*, può essere usata per realizzare la struttura dati (astratta) tabella.

Tabella $m \times n$: disposizione rettangolare di m righe ed n colonne, dove ciascuna colonna contiene elementi (tutti) di tipo t_j ed è individuata da un nome A_j (*attributo*), con $1 \leq j \leq n$, mentre ciascuna riga è individuata da un intero i (*indice*), con $0 \leq i < m$. □

Graficamente:



Ciascuna riga è una n-upla (o *record*), $A_1, \ldots, A_n$ sono gli *attributi* della n-upla e $t_1, \ldots, t_n$ i *domini* di ciascun attributo.

Più formalmente, una tabella T può essere definita come un sottoinsieme del prodotto cartesiano dei domini $t_1, \ldots, t_n$, cioè:

$$T \subseteq t_1 \times t_2 \times \ldots \times t_n = \{\langle a_1, a_2, \ldots, a_n \rangle : a_1 \in t_1, \ldots, a_1 \in t_n\}$$

ovvero una relazione n-aria su $t_1, \ldots, t_n$. Si noti che la cardinalità dell'insieme T corrisponde al numero di righe nella tabella.

Esempio 4.31 *Tabella* esami:

	materia	*voto*	*lode*	*data_es*
0				
1				
	...	...	...	...
49				

dove: l'attributo materia *è di tipo stringa,* voto *di tipo numero intero,* lode *di tipo booleano, e* data_es *di tipo* data.

Tabella amici:

	nome	*cognome*	*indirizzo*	*tel*	*data_nascita*
0					
1					
	...	...	...	...	...
99					

dove: gli attributi nome, cognome *e* indirizzo *sono di tipo stringa,* tel *di tipo numero intero e* data_nascita *di tipo* data.

La struttura dati astratta tabella $m \times n$ può essere realizzata in C++ tramite un *array* di *struct* (struttura dati concreta) nel modo seguente:

```
struct n_upla
    {t1 A1;
     t2 A2;
     ...
     tn An;
    };
n_upla T[m];
```

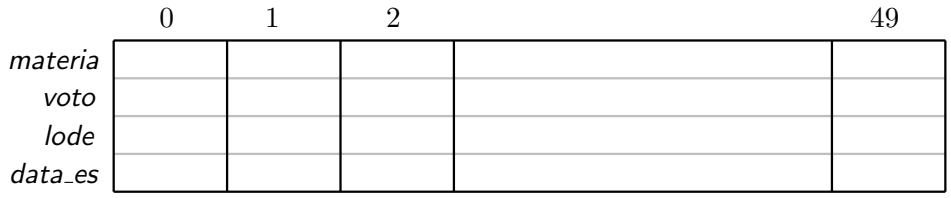
dichiara un *array* di nome `T` costituito da `m` elementi di tipo `n_upla`.

Esempio 4.32 (Realizzazione tabella esami in C++)

```
struct dati_esame
    {char materia[100];
     int voto;
     bool lode;
     data data_esame;
    };
dati_esame esami[50];
```

in cui si assume che: la lunghezza massima del titolo dell'esame sia 100 *caratteri, il numero massimo di esami (= righe della tabella) sia* 50*, e* `data` *sia il solito tipo definito da utente che realizza la data del giorno (tripla* $\langle g, m, a \rangle$*).*

Rappresentazione grafica come array:



Mostriamo ora programma completo C++ che realizza e gestisce una tabella.

Esempio 4.33 (Gestione tabella esami)

Problema. Leggere una sequenza di dati relativi agli esami sostenuti da uno studente:

- *titolo esame*
- *voto (in trentesimi, tra* 18 *e* 30*)*
- *eventuale lode*
- *data esame*

fino a leggere (un esame con titolo) "`stop`". Memorizzare i dati letti in un'opportuna tabella `esami`, stampare la tabella letta, e quindi effettuare uno o più elaborazioni sulla tabella (ad es., ricerca dell'esame con il voto massimo) e stampare i relativi risultati.

```
#include <iostream>
#include <cstring>

using namespace std;

const int dim_tabella = 50;  // capacita' max della tabella esami
const int dim_titolo = 100;  // capacita' max del nome dell'esame
struct data {
     int giorno;
     int mese;
     int anno;
};
struct dati_esame {
     char materia[dim_titolo];
     int voto;
     bool lode;
     data data_esame;
};
int main()
     {
      dati_esame esami[dim_tabella];  // tabella esami
      int i=0;  // i: indice array esami;
      do {
         char titolo[dim_titolo];
         cout << "\nDai nome dell'esame (stop per smettere): ";
         cin.getline(titolo,dim_titolo);
         if (strcmp(titolo,"stop")==0) break;
         else strcpy(esami[i].materia,titolo);

         int voto;
         cout << "Dai voto in trentesimi: ";
         do cin >> voto; while (voto < 0 || voto > 30);
         esami[i].voto = voto;

         char lode;
         if (esami[i].voto == 30) {
            cout << "Lode? (sn) ;/
            do cin >> lode; while (lode != 'n' && lode != 's');
            if (lode == 'n') esami[i].lode = false;
         else esami[i].lode = true;
         }

         data data_esame;
         cout << "Dai data esame (g m a): ";
         cin >> data_esame.giorno
             >> data_esame.mese
             >> data_esame.anno;
         // aggiungere controlli su validita' data !!
         esami[i].data_esame = data_esame;
```

```
    i++;
    cin.ignore(256,'\n');  // elimina "a capo" successivo
  }                        // ultimo intero immesso
  while(i < dim_tabella);

  int n=i;
  // stampa della tabella dei voti
  for (int i=0; i<n; i++) {
     cout << "\nMateria: " << esami[i].materia << endl;
     cout << "Voto: " << esami[i].voto;
     if (esami[i].voto == 30 && esami[i].lode) cout << " con lode";
     cout << endl;
     cout << "Data: " << esami[i].data_esame.giorno << '.'
                      << esami[i].data_esame.mese << '.'
                      << esami[i].data_esame.anno << endl;
  }
  cout << endl;

  // calcolo del massimo (senza tener conto delle lodi)
  int max=0;
  for (int i=0; i<n; i++)
     if (esami[i].voto > max) max = esami[i].voto;
  cout << "Il voto massimo e' " << max << endl << endl;

  return 0;
}
```

4.8 Ancora sui tipi di dato (IN PREPARAZIONE)

4.8.1 Nome di un tipo

4.8.2 Utilità dei tipi

4.9 *Domande per il Capitolo 4*

1. Dire cosa è e da cosa è caratterizzato, in generale, un tipo di dato strutturato.
2. Indicare almeno tre costruttori di tipo del C++.
3. Indicare il nome di almeno tre strutture date astratte realizzabili tramite array.
4. Cosa è un array (in generale)? Come lo si dichiara in C++?
5. Qual è il (nuovo) tipo strutturato introdotto dalla dichiarazione

   ```
   int A[10];
   ```
   ?
6. Mostrare la dichiarazione di un array di 20 caratteri con l'inizializzazione (tramite *lista di inizializzatori*) dei suoi primi 5 elementi con le prime cinque lettere dell'alfabeto.
7. Qual è la dimensione dell'array introdotto dalla seguente dichiarazione:

   ```
   float A[];
   ```
   ?

   E di quello introdotto dalla dichiarazione:

   ```
   float B[] = {-2.1, +2.3};
   ```
   ?
8. Cosa si intende per *operazione di selezione* su un array? Quali sono le sue caratteristiche e qual è la forma sintattica e il significato in C++?
9. Se `A` è un array di interi e `i` una variabile intera, qual è la differenza, in generale, tra le seguenti due espressioni

   ```
   A[i+1]
   ```

   ```
   A[i]+1
   ```
   ?
10. Se `A` è un array di `float` e `i` è una variabile intera, qual è il tipo della seguente espressione

    ```
    A[i+1]
    ```
    ?
11. Se `A` è un array di interi e `i` una variabile intera, qual è la differenza di significato tra `A[i]` posto a destra di un assegnamento e `A[i]` posto a sinistra?

12. Cosa accade eseguendo il seguente frammento di codice C++

```
int A[20]
int i = 19
A[i] = A[i+1] ?
```

13. Se `A` è un array di k interi, allocato a partire dall'indirizzo di memoria b, come si calcola l'indirizzo di memoria del generico elemento `A[i]` (si assuma `sizeof(int)` $= 4$)?

14. Quali operazioni sono applicabili ad un intero array (cioè non ai singoli elementi dell'array)? E quali ad intere *struct*?

15. Cosa si intende con array bidimensionale (definizione generale)? Come lo si dichiara in C++?

16. Illustrare con un disegno l'allocazione in memoria del seguente array bidimensionale:

```
char B[3][2] = {{'b','p'},{'f','v'},{'d','t'}};
```

17. Qual è in C++ la forma sintattica e il significato dell'operazione di selezione di un elemento in un array bidimensionale t `B`$[m]$`[`n`];`?

18. Cosa si intende con array semi-dinamici? A quali limitazioni d'uso sono soggetti in C++?

19. Che comportamento provoca in generale l'esecuzione del seguente frammento di codice C++ (si supponga di leggere in `m` il valore 5):

```
int m = 0;
int V[m];
cin >> m;
V[2] = 3;
```

20. Come può essere realizzata in C++ una matrice M di $m \times n$ elementi di tipo t?

21. Si consideri una matrice quadrata *10 x 10* realizzata tramite l'array `int M[10][10]`. Calcolare la somma `d` di tutti gli elementi sulla diagonale principale della matrice rappresentata da `M`.

22. Si consideri una matrice quadrata *10 x 10* realizzata tramite l'array `int M[10][10]`. Calcolare la somma `c` di tutti gli elementi della prima colonna.

23. Cosa è una stringa e quali sono tipiche operazioni su stringhe (struttura dati astratta)?

24. Come viene realizzata in C++ la struttura dati (astratta) stringa mediante array? (stringhe in "stile C")?

25. Mostrare con un disegno la memorizzazione concreta della stringa

```
char s[15] = ciao mondo;
```

26. Dichiarare una stringa `s` ("stile C"), di dimensione massima 10, ed inizializzarla con la stringa vuota. Mostrare quindi, con un disegno, la sua memorizzazione interna.
27. Qual è la differenza (in termini di rappresentazione interna concreta) tra le costanti `'a'` e `"a"` in C++?
28. Quali operazioni primitive fornisce il C++ per operare su stringhe ("stile C")?
29. Quali sono possibili limitazioni nell'uso dell'operatore `>>` per la lettura di una stringa "stile C"?
30. Qual è la forma sintattica ed il funzionamento della funzione di libreria `getline`?
31. Indicare e descrivere brevemente alcune funzioni della libreria `cstring` per operare su stringhe "stile C".
32. Cosa è una *struct* (in generale)? Come lo si dichiara in C++?
33. Quali sono le differenze (sintattiche e semantiche) tra la dichiarazione di un tipo *struct* e di una variabile di tipo *struct*?
34. Quali delle seguenti tre dichiarazioni comportano allocazione di memoria?

```
struct persona {char nome[32];
                int eta;};
persona p1 ;
persona p2 = {"mario",35};
```

35. Se `s` è la *struct* definita dalla seguente dichiarazione

```
struct s {int a; char b[10];}
```

come si accede all'elemento di indice 0 del campo `b` di una variabile `x` di tipo `s`?

36. Mostrare le dichiarazioni necessarie a realizzare in C++ una tabella `T` con n colonne, di nome `C1`, `C2`, ..., `Cn` e tipi `t1`, `t2`, ..., `tn`, e al più k righe, tramite un array di *struct*.
37. A cosa serve il costrutto `typedef` del C++?
38. Definire un nuovo tipo array di interi, di dimensione 10, e associargli mediante `typedef` il nome `vettore10`.
39. Indicare almeno tre vantaggi dell'uso dei tipi in un linguaggio di programmazione.