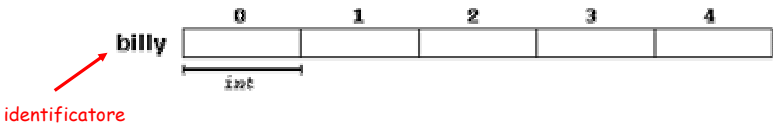
Strutture Dati

- Le strutture dati sono entità che permettono di memorizzare i dati in modo organizzato e funzionale a particolari compiti computazionali.
- Il loro utilizzo corretto spesso permette di realizzare algoritmi efficienti e veloci.
- Noi vedremo come alcune strutture dati possono essere definite ed utilizzate in C++, ed in particolare tratteremo i costrutti linguistici che riguardano:
  - Array;
  - Stringhe di caratteri;
  - Strutture;
  - Tipi definiti dall'utente.

87

Array

- Gli array sono sequenze di variabili dello stesso tipo che vengono situate consecutivamente nella memoria ed alle quali è possibile accedere usando uno stesso nome (identificatore) a cui viene aggiunto un indice.
- Ad esempio, possiamo memorizzare 5 valori di tipo **`int`** senza bisogno di dichiarare cinque diverse variabili con cinque diversi identificatori: è sufficiente dichiarare un array di cinque elementi dello stesso tipo **`int`** con un solo identificatore:



identificatore

ogni cella rappresenta un elemento dell'array. Gli elementi sono numerati da **`0`** a **`4`** in quanto, in un array, l'indice del primo elemento è sempre **`0`** (e non **`1`**).

88

Array

- Come tutte le variabili anche gli array devono essere dichiarati prima di poterli usare.
- Un esempio di dichiarazione di un array in C++ è:

```
tipo nome [dimensione];
```

- dove **tipo** è il tipo degli elementi (**int, float** …) detto anche tipo base dell'array , **nome** è un identificatore e **dimensione**, che deve essere racchiuso tra parentesi quadre **[]**, è la dimensione, ossia il numero di elementi, dell'array.
- La dichiarazione dell'array **billy** è:

```
int billy [5];
```

ATTENZIONE: Il campo **dimensione** deve essere un valore costante in quanto gli array sono blocchi di memoria di dimensione prefissata ed il compilatore deve conoscere esattamente quanta memoria serve per l'array prima che il programma venga eseguito.

89

Array: inizializzazione

- Come per le variabili semplici, anche per gli array è possibile specificare un valore iniziale. Ad esempio, con la dichiarazione:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

l'array viene inizializzato come segue:

	0	1	2	3	4
billy	16	2	77	40	12071

- Il numero di valori usati per l'inizializzazione (quelli posti tra le parentesi grafe **{}**) deve essere esattamente uguale alla dimensione dell'array.
- In C++ è possibile anche usare la notazione:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

ed in questo caso viene assunto implicitamente come dimensione dell'array il numero di valori della lista di inizializzazione.

ATTENZIONE !

In un array senza inizializzazione il valore iniziale dei suoi elementi risulta indeterminato (i bit della memoria riservata per l'array conservano i valori lasciati dai programmi precedenti).

90

Array: accesso ai valori

- In ogni punto del programma in cui un array risulta visibile possiamo accedere individualmente ad uno degli elementi dell'array per leggerlo o modificarlo esattamente come esso fosse una normale variabile. Il formato è il seguente:

name[index]

- Proseguendo l'esempio dell'array **billy** di 5 elementi di tipo **int**, i nomi mediante i quali possiamo accedere a ciascun elemento dell'array sono quelli indicati sopra le singole celle nella figura seguente:

- Ad esempio, se vogliamo memorizzare il valore 75 nel terzo elemento di **billy** possiamo usare l'assegnamento:

```
billy[2] = 75;
```

oppure, per copiare il valore del terzo elemento nella variabile **a** possiamo usare:

```
a = billy[2];
```

Si comporta come una variabile di tipo **int**

91

Array: accesso ai valori

- Occorre notare i due diversi usi delle parentesi quadre [ ] con gli array:
  - nella dichiarazione di un array esse sono usate per indicare la dimensione dell'array ;
  - in tutti gli altri contesti esse vengono usate per specificare un indice per individuare un particolare elemento dell'array.

```
int billy[5];      // dichiarazione di un nuovo array di 5 elementi
billy[2] = 75;     // accesso ad un elemento particolare
                   // dell'array: quello di indice 2.
```

- Altre possibili operazioni con gli array sono:

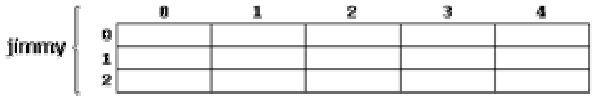
```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// esempio con gli array
#include <iostream.h>

int billy [] = {16, 2, 77, 40, 12071};
int n, risultato=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    risultato += billy[n];
  }
  cout << risultato;
  return 0;
}
```

```
12206
```

Array multidimensionali

- Un array multidimensionale si può pensare come un array di array di array di ....
- Ad esempio, un array bidimensionale si può pensare come una tabella bidimensionale i cui elementi appartengono tutti allo stesso *tipo*.



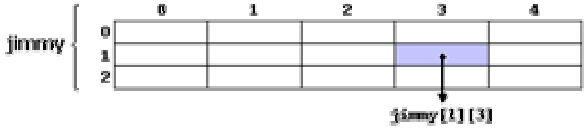
- **jimmy** è un array bidimensionale di 3 per 5 valori di tipo **int** .
- Esso si può pensare come un array di 5 elementi, ciscuno dei quali è a sua volta un array di 3 elementi (le colonne). Si dichiara come segue:

```
int jimmy [3][5];
```

e, per riferirsi all'elemento nella seconda riga e nella quarta colonna si usa la notazione:

```
jimmy[1][3]
```

ricordiamo che gli indici degli array iniziano sempre con 0.



93

Array multidimensionali

- Gli array multidimensionali possono avere più di due indici (due dimensioni). Ad esempio:

```
char secolo [100][365][24][60][60];
```

che però richiede memoria per un valore **char** per ogni secondo contenuto in un secolo, più di 3 miliardi di **char** ! Il che richiede più di 3 gigabytes di memoria RAM.

- Gli elementi di un array multidimensionale sono memorizzati nella RAM uno di seguito all'altro come per gli array semplici:

```
int jimmy [3][5];    è equivalente a    int jimmy [15];
```

con l'unica differenza che il compilatore gestisce per noi la suddivisione in righe e colonne:

```
// array multidimensionale
#include <iostream.h>
#define COLONNE 5
#define RIGHE 3

int jimmy [RIGHE][COLONNE];
int n,m;
int main ()
{
  for (n=0;n<RIGHE;n++)
    for (m=0;m<COLONNE;m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
  return 0;
}
```

```
// array pseudo-multidimensionale
#include <iostream.h>
#define COLONNE 5
#define RIGHE 3

int jimmy [RIGHE * COLONNE];
int n,m;
int main ()
{
  for (n=0;n<RIGHE;n++)
    for (m=0;m<COLONNE;m++)
    {
      jimmy[n * COLONNE + m]=(n+1)*(m+1);
    }
  return 0;
}
```

94

Array multidimensionali

```
// array multidimensionale
#include <iostream.h>
#define COLONNE 5
#define RIGHE 3

int jimmy [RIGHE][COLONNE];
int n,m;
int main ()
{
  for (n=0;n<RIGHE;n++)
    for (m=0;m<COLONNE;m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
  return 0;
}
```

```
// array pseudo-multidimensionale
#include <iostream.h>
#define COLONNE 5
#define RIGHE 3

int jimmy [RIGHE * COLONNE];
int n,m;
int main ()
{
  for (n=0;n<RIGHE;n++)
    for (m=0;m<COLONNE;m++)
    {
      jimmy[n * COLONNE + m]=(n+1)*(m+1);
    }
  return 0;
}
```

- Entrambi i programmi assegnano i seguenti valori al blocco di memoria riservato per **jimmy**:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

- Notate l'uso della direttiva **#define** per facilitare ridimensionamenti dell'array.

95

Array come parametri

- Potremmo voler passare un array come parametro ad una funzione.
- In C++ non è possibile passare come parametro ad una funzione il valore di un array (ossia l'insieme dei valori di tutti i suoi elementi).
- Però possiamo passare come parametro l'indirizzo dell'array.
- Per indicare che un argomento di una funzione rappresenta un parametro di tipo array basta scrivere il tipo degli elementi dell'array (il tipo base dell'array) seguito da una coppia di parentesi quadre [].
- Ad esempio la funzione:

```
void procedura (int arg[])
```

aspetta un argomento **arg** di tipo "Array di **int** ". Per passare alla funzione l'array:

```
int mioarray [40];
```

è sufficiente scrivere una chiamata della funzione del tipo:

```
procedura (mioarray);
```

96

Array come parametri

Esempio di passaggio di un array come parametro

```
// array come parametri
#include <iostream.h>

void stampaarray (int arg[], int lunghezza)
{
 int n;
  for (n=0; n<lunghezza; n++)
    cout << arg[n] << " ";
  cout << "\n";
}

int main ()
{
  int primoarray[] = {5, 10, 15};
  int secondoarray[] = {2, 4, 6, 8, 10};
  stampaarray (primoarray,3);
  stampaarray (secondoarray,5);
  return 0;
}
```

```
5 10 15
2 4 6 8 10
```

Come si vede il primo argomento (**int arg[]**) ammette qualsiasi array con tipo base **int**, indipendentemente dalla sua dimensione.
Il secondo argomento segnala alla funzione quale sia la dimensione dell'array passato come primo argomento. La dimensione dell'array è necessaria perché il ciclo **for** possa stampare il giusto numero di elementi.

97

Array come parametri

- In una dichiarazione di funzione si possono anche usare array multidimensionali come argomenti.
- Ad esempio, la forma dell'argomento per un array tri-dimensionale è:

  `tipo_base [][d2][d3]`

  in cui devono essere specificate tutte le dimensioni ad esclusione della prima.
- Tale argomento ammette un qualsiasi array tridimensionale avente seconda e terza dimensione **d2** e **d3** prefissate e prima dimensione qualsiasi.
- Ad esempio:

  `void procedura (int mioarray[][3][4])`
- La ragione per cui **d2** e **d3** devono essere fissate è che il compilatore, per determinare la posizione in memoria dell'elemento **`mioarray[i][j][k]`** usa la formula **`i*3*4+j*4+k`** .

98

Esercizi su Array

- Esercizio 1:
  Scrivere un programma che riceve da tastiera il punteggio raggiunto da sei studenti e ne determina il maggiore, il minore e la media.

- Esercizio 2:
  Riscrivere il programma precedente dove il codice di ricezione dei dati in ingresso è raccolto in una funzione con il seguente prototipo:

```
void leggidatiingresso (int dati[], int lunghezza);
```

99

Altro esercizio su Array

- Esercizio 3:
  Scrivere un programma che riceve da tastiera i valori associati a due matrici A (di dimensione n x p) e B (di dimensione p x m), le cui dimensioni sono definite nel programma attraverso una direttiva #define, e stampi la matrice prodotto C=AB. Vi ricordo la formula per il prodotto di matrici:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

100

Stringhe di caratteri

- Finora abbiamo usato le stringhe di caratteri come costanti ma non abbiamo ancora visto variabili che possano contenerle.
- In C++ non vi è un tipo di variabile predefinito in grado di memorizzare delle stringhe di caratteri. Dobbiamo usare degli array di caratteri.
- Vediamo ora il trattamento standard (stile C) delle stringhe come array di caratteri.
  Il seguente array:

```
char jenny [20];
```

  può memorizzare una stringa di al più 20 caratteri. Possiamo rappresentarlo come:

- Non è necessario usare tutti e 20 i caratteri dell'array. Per questo motivo occorre prevedere una indicazione del punto in cui termina la stringa. Per convenzione tale punto viene indicato dal carattere nullo che si può scrivere sia **0** sia **'\0'**:

101

Stringhe: inizializzazione

- Per inizializzare una stringa di si può usare la stessa notazione usata per gli array:

```
char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

- In alternativa si può usare anche una stringa costante:

```
char mystring [] = "Hello";
```

- In entrambi i casi la dimensione dell'array **mystring** è di 6 elementi di tipo **char**: i 5 caratteri di **Hello** e il carattere nullo finale (**'\0'**).
- **Attenzione**: una stringa costante può essere usata per dare un valore iniziale ad una variabile di tipo array di **char** soltanto al momento della dichiarazione dell'array, ossia in fase di inizializzazione.
- Assegnamenti quali:

```
mystring = "Hello";
mystring[] = "Hello";
```

  non sono permessi, come non è permesso:

```
mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

102

Stringhe: assegnamenti

- Siccome in un assegnamento l'lvalore può essere soltanto un elemento di un array e non l'intero array, per assegnare una stringa di caratteri ad un array di **char** dobbiamo scrivere:

```
mystring[0] = 'H';
mystring[1] = 'e';
mystring[2] = 'l';
mystring[3] = 'l';
mystring[4] = 'o';
mystring[5] = '\0';
```

- Poiché questo non è molto pratico la libreria standard **cstring** (che si può includere con la direttiva **#include <string.h>**) contiene la definizione di un certo numero di funzioni, quali **strcpy** (**str** ing **co** **py**) che si può richiamare nel seguente modo:

```
strcpy (string1, string2);
```

l'effetto è copiare il contenuto di ***string2*** in ***string1***. ***string2*** può essere sia un array sia una stringa costante, il che ci permette di assegnare la stringa costante **"Hello"** all'array di caratteri **mystring** usando la seguente notazione:

```
strcpy (mystring, "Hello");
```

103

Stringhe: assegnamento

Esempio di assegnamento realizzato utilizzando la libreria standard

```
// assegnamento a stringhe
#include <iostream.h>
#include <string.h>

int main ()
{
  char stMyName [20];
  strcpy (stMyName,"J. Soulie");
  cout << stMyName;
  return 0;
}
```

```
J. Soulie
```

Esempio di assegnamento realizzato "in casa": cioè definendo una nostra funzione di copia

```
// assegnamento a stringhe
#include <iostream.h>

void setstring (char stOut [], char stIn [])
{
  int n=0;
  do {
    stOut[n] = stIn[n];
  } while (stIn[n++] != '\0');
}

int main ()
{
  char stMyName [20];
  setstring (stMyName,"J. Soulie");
  cout << stMyName;
  return 0;
}
```

```
J. Soulie
```

104

Stringhe: assegnamenti

- Un altro modo per assegnare un valore ad un array di caratteri è quello di usare direttamente il flusso di input **cin**.
- Nella libreria **iostream** è infatti definita una funzione **getline** il cui prototipo è:

```
cin.getline ( char buffer [], int length, char delimiter = '\n');
```

dove **buffer** è l'array di caratteri in cui memorizzare l'input, **length** è la dimensione dell'array stesso e **delimiter** è il carattere usato per indicare la fine dell'input e per il quale è previsto il carattere nuova linea (**'\n'**) come valore di default.

```
// uso di cin.getline
#include <iostream.h>

int main ()
{
  char buff [100];
  cout << "Come ti chiami? ";
  cin.getline (buff,100);
  cout << "Salve " << buff << ".\n";
  cout << "La tua squadra preferita? ";
  cin.getline (buff,100);
  cout << "L'" << buff << " piace anche a me.\n";
  return 0;
}
```

```
Come ti chiami? Juan
Salve Juan.
La tua squadra preferita? Inter
L'Inter piace anche a me.
```

Stringhe: assegnamenti

- Si può anche usare l'operatore di estrazione (>>) per leggere delle stringhe da **cin** :

```
cin >> buff;
```

che funziona ma con le seguenti limitazioni che **cin.getline** non ha:

  - si possono leggere soltanto parole e non intere frasi in quanto l'operatore di estrazione usa come delimitatore qualsiasi occorrenza di un carattere invisibile (spazio, tabulazione, nuova linea, ritorno carrello).
  - non si può specificare la dimensione dell'array il che rende instabile il programma nel caso in cui l'input sia una parola più lunga della dimensione dell'array.

- Altre funzioni che operano su stringhe sono definite nella libreria **cstring** (**string.h**).

106

Esercizi su stringhe

- Esercizio 4:
  Scrivere una funzione che date due stringhe (di lunghezza massima M) restituisca la stringa risultante dalla concatenazione della prima con la seconda:

```
void concatenastringhe(char st1[], char st2[], char stOut[])
```

- Esercizio 5:
  Scrivere una funzione che date due stringhe (di lunghezza massima M) restituisca la stringa che alterna i caratteri della prima con quelli della seconda:

```
void mischiastringhe(char st1[], char st2[], char stOut[])
```

Esempi:
dati "**ancora**", "**cane**" restituisce "**acnacnoera**"
"**amo**", "**salmone**" restituisce "**asmaolmone**"

107

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[])
```

108

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.

109

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.

110

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`**

111

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
```

114