



Dipartimento
di Fisica
e Astronomia
Galileo Galilei

1222·2022
800
ANNI



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sperimentazioni di Fisica I

mod. A – Lezione 8

Array, Stringhe e Vettori (CAP. 3)

*Dipartimento di Fisica e Astronomia “G. Galilei”,
Università degli Studi di Padova*

Array, Stringhe e Vettori

Lezione 8:

Parte 1. Array

Array (Statici) in C++

- ✓ permette di immagazzinare più di un elemento dello stesso tipo
- ✓ e di accedere ad un singolo elemento dell'array specificando la sua posizione
- ✗ hanno una **dimensione fissata** (i.e. numero di elementi)
- ✗ **non è possibile aggiungere o eliminare elementi** durante l'esecuzione del programma

Sintassi di dichiarazione di un array

Tipo di dato numero di elementi

↓ ↓

TypeName arrayName[arraySize];

 ↑

 nome dell'array

Esempio

```
int months[12];      // array di tipo int con 12 elementi
```

Creazione di Array (Statici)

Creo un array con 4 elementi:

- ✓ primo elemento `seasons[0]`
- ...
- ✓ ultimo elemento `seasons[3]`

```
int seasons[4];  
[0]  
[1]  
[2]  
[3]
```

```
TypeName arrayName [arraySize]
```

`arraySize` può essere:

una costante intera

una espressione costante

```
const int size = 4; double wine[8*32+size];  
double wine[size];
```

NON è lecita una variabile il cui valore cambia run-time

```
int dim = 3;  
double boxes[dim];
```

A Cosa Servono gli Array?



Inizializzazione di Array (Statici)

- ✓ Gli elementi di un array non sono automaticamente inizializzati
- ✓ esistono svariati modi per **inizializzarli** :

```
const unsigned size = 3;    // La dimensione e' costante
int a[size] = {0, 1, 3};    // a[0] = 0, a[1] = 1, a[2] = 3

int a2[] = {0, 1, 3};      // Un array di dimensione 3
int a3[5] = {0, 1, 3};     // equivalente a
                           // a3[] = {0, 1, 3, 0, 0}

int a4[2] = 0, 1, 3;       // ERRORE: troppi inizializzatori
int a5[];                  // ERRORE: dimensione sconosciuta
```

✗ ATTENZIONE: No COPY or ASSIGNMENT

```
int b[] = a;               // ERRORE: non possibile inizializzare
                           // un array con un altro

int c[size];
c = a;                     // ERRORE: non possibile assegnare
                           // un array con un altro
```

Puntatori ed Array

- Tutte le volte che usiamo l'operatore *address-of* & di fronte ad un oggetto, otteniamo un *puntatore all'oggetto*
- Gli elementi di un array sono oggetti : con l'operatore *subscript* [] si ottiene l'oggetto alla locazione indicata dall'indice

```
const int size = 3;
string num[size] = {"uno", "due", "tre"};

string *ps = & num[0]; // punta al primo elemento
ps = num;              // equivalente, il nome di un array
                       // e' un puntatore costante al primo
                       // elemento dell'array

string *first = num;   // punta al primo elemento
string *last  = num + size - 1; // punta all'ultimo elemento
while (first != last) {
    cout << *first << endl;
    ++first;
}
cout << *last << endl; // stampiamo anche l'ultimo
```

begin () ed end ()

- Gli array standard del C++ non permettono un controllo sulla validità dell'indice dell'elemento al quale si accede
- Il C++ fornisce due funzioni `begin` e `end` che forniscono le stesse funzionalità degli equivalenti membri dei contenitori (per es `string` e `vector`)

```
int ia[] {0,1,2,3,4,5,6,7,8,9};

int *first = begin(ia); // punta al primo elemento
int *last = end(ia);   // punta all'elemento
                       // successivo all'ultimo

while (first != last) {
    cout << *first << endl;
    ++first;
}
```


Array Dinamici (CAP. 12)

- ✓ Gli array sono contenitori statici : è necessario conoscere la dimensione dell'array a *compile-time* e non è possibile aggiungere o togliere elementi *run-time*
- ✓ gli operatori *new/delete* permettono di allocare/liberare memoria per immagazzinare oggetti *run-time*
- ✓ possiamo utilizzarli per creare array dinamici

```
TypeName * nomePuntatore = new TypeName[arraySize]
```

```
double * pd = new double; // crea spazio per un double  
int * pia = new int[10]; // crea un array con 10 int
```

```
for (int i=0; i<10; i++)  
    cin >> pia[i];
```

```
delete pd; // elimina il double  
delete [] pia; // distrugge tutto l'array
```

Il Contenitore array

- ✓ Il contenitore generico `array<class T, size_t N>`, definito nel file header `<array>`, permette di creare un numero prefissato di elementi, grazie al parametro `N`
- ✓ completamente compatibile, a livello binario, con gli array standard del C :
 - ✓ fornisce accesso casuale agli elementi con l'operatore `[]`
 - ✓ rispetta il vincolo di contiguità di memoria degli elementi `a[i]`

```
array<double, 12> periodo;    // array di 12 double
```

è equivalente a

```
double periodo[12];
```

- il vantaggio, rispetto agli array del C, è di poter usare funzioni specializzate direttamente implementate come metodi

```
#include <array>
int main() {
    array<int, 4> a1 = {1, 2, 3, 4};
    array<int, 4> a2;
    // Scambiamo gli array
    a2.swap(a1);
    a1.fill(-1);
}
```

Array Multidimensionali

Dichiarazione di array multidimensionali:

```
int ia[3][4]; // array of size 3; each element is an array of ints of size 4
// array of size 10; each element is a 20-element array whose elements are arrays of 30
ints
int arr[10][20][30] = {0}; // initialize all elements to 0
```

Inizializzazione di array multidimensionali:

```
int ia[3][4] = { // three elements; each element is an array of size 4
    {0, 1, 2, 3}, // initializers for the row indexed by 0
    {4, 5, 6, 7}, // initializers for the row indexed by 1
    {8, 9, 10, 11} // initializers for the row indexed by 2
};

// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

Array, Stringhe e Vettori

Lezione 8:

Parte 2. Stringhe

La Classe `string`

- Una `string` è una sequenza variabile di caratteri
- È definita nel namespace `std`
- Per creare/manipolare oggetti della classe `string` è necessario scrivere

```
#include <string>
using std::string;
```

- definizione e inizializzazione di una stringa:

```
string s1;
```

← inizializzazione di default:
s1 è una stringa vuota

```
string s2 = s1;
```

← s2 è una copia di s1

```
string s3 = "Cpp";
```

← s3 è una copia di una stringa letterale

```
string s4(10, 'S');
```

← s4 è "SSSSSSSSSS"

Inizializzazione di Stringhe

- Il C++ permette **svariate forme di inizializzazione**
- Inizializzando una variabile **con l'operatore =**, chiediamo al compilatore una **inizializzazione con copia** dell'oggetto
- **senza l'operatore =**, si usa invece la **inizializzazione diretta**

```
string s1; // Init default: stringa vuota
```

```
string s2(s1); // Init diretta: s2 copia di s1
```

```
string s2 = s1; // Init di copia, equiv. a s2(s1)
```

```
string s1("Prova"); // Inizializzazione diretta
```

```
string s1 = "Prova"; // Inizializzazione di copia
```

Letture e Scrittura di Stringhe

Utilizzo della libreria iostream come per int, double

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "Inserire una stringa: "

    std::string s;

    std::cin >> s;

    std::cout << "Stringa: \""
              << s << "\""
              << std::endl;
    return 0;
}
```

- compiliamo ed eseguiamo il programma

```
$ g++ string1.cxx
$
$ ./a.out
Inserire una stringa: Ciao
String: "Ciao"
$
```

```
$ ./a.out
Inserire una stringa: Ciao a tutti
String: "Ciao"
$
```

**std::cin legge caratteri fino a quando non
incontra un carattere whitespace**

Numero Variabile di Stringhe

Scriviamo un programma per leggere un numero sconosciuto di stringhe

```
#include <iostream>
#include <string>

int main()
{
    std::string word;
    while (std::cin >> word) {
        std::cout << "Stringa: \""
                  << word << "\"\n"
                  << std::endl;
    }
    return 0;
}
```

```
$ g++ -Wall -o string1 string1.cpp

$ ./string1

$ ./string1
Prima riga di input
Stringa: "Prima"
Stringa: "riga"
Stringa: "di"
Stringa: "input"

Seconda
Stringa: "Seconda"

$
```

Il while controlla la validità della stream dopo ogni lettura. Se ci sono errori (per esempio End-Of-File), esce dal ciclo

Lettura di una Riga Intera

- Scriviamo un programma per leggere un numero sconosciuto di righe (riga = caratteri fino a '`\n`')

```
#include <iostream>
#include <string>

int main()
{
    std::string line;

    // Read input line until EOF
    while (std::getline(std::cin, line))
    {
        std::cout << "Line: \""
                  << line << "\"\n"
                  << std::endl;
    }

    return 0;
}
```

```
$ g++ -Wall -o string2 string2.cpp

$ ./string2

Prova
Line: "Prova"

Uno Due Tre
Line: "Uno Due Tre"

$
```

`getline()` legge tutti i caratteri
fino al carattere `newline`

Sintassi: `getline (cin, nome_della_stringa);`

empty () e size ()

La funzione `empty ()` ritorna un `bool` che indica se la stringa è vuota

```
std::string line;
while (std::getline(std::cin, line) ) {
    if (!line.empty()) {
        std::cout << line << std::endl;
    }
}
```

La funzione `size ()` ritorna la lunghezza della stringa: il numero di caratteri in essa contenuti.

```
std::string line;
while (std::getline(std::cin, line) ) {
    if (line.size() < 80) {
        std::cout << line << std::endl;
    }
}
```

Concatenazione di Stringhe

L'operatore `+` permette di concatenare due stringhe (sum)

L'operatore composto `+=` attacca l'operando a destra alla stringa di sinistra (append)

```
string s1 = "hello, "; string s2("World!");
```

```
string s3 = s1 + s2; ← ok: hello, World!
```

```
s1 += s2; ← ok: hello, World!
```

È possibile concatenare oggetti stringhe con stringhe costanti

```
string s1 = "hello", s2("World");
```

```
string s3 = s1 + ", " + s2 + "!"; ← ok: hello, World!
```

Mescolando stringhe e costanti ⇒ **uno dei due op. di tipo string**

```
string s4 = s1 + "!!!"; ← ok: hello!!!
```

```
string s5 = "hello " + "Sky"; ← err: no string operand
```

```
string s6 = s1 + ", " + "Sky"; ← ok: (s1 + ",") + "Sky"
```

```
string s7 = "hello" + ", " + s2; ← err: can't add string literals
```

range for

- A volte è necessario **accedere ai singoli caratteri** della stringa
- L'approccio migliore, per accedere ad ogni singolo carattere, è di usare una **istruzione range for**

Sintassi:

```
for (declaration : expression)
    statement;
```

C++11

- Una **string** rappresenta una **sequenza di caratteri**, pertanto possiamo usarla come **expression** nel **range for**

```
std::string line("0123456789abcdef");
for (auto c : line) {
    std::cout << c;
}
```

C++11

`auto` dirà al compilatore di determinare il tipo di `c`, che nel nostro caso sarà `char`

Esempio di Utilizzo

```
#include <iostream>
#include <string>
#include <cctype>

int main()
{
    std::string s;
    int alpha_cnt = 0;
    int digit_cnt = 0;
    int other_cnt = 0;

    std::getline(std::cin, s);
    for (auto c : s) {
        if (isalpha(c))
            ++alpha_cnt;
        else if (isdigit(c))
            ++digit_cnt;
        else
            ++other_cnt;
    }

    std::cout << "Letters: "
               << alpha_cnt
               << " Numbers: "
               << digit_cnt
               << " Others: "
               << other_cnt
               << std::endl;

    return 0;
}
```

```
$ g++ -W -Wall -std=c++0x str_cnt.cxx
```

```
$ ./a.out
```

```
Test del programma 12345 &^#%
```

```
Letters: 16 Numbers: 5 Others: 8
```

Funzioni ctype

`isalnum(c)` true se `c` è lettera o numero

`isalpha(c)` true se `c` è una lettera

`isdigit(c)` true se `c` è una cifra numerica

`iscntrl(c)` true se `c` è un carattere di controllo

`ispunct(c)` true se `c` è un carattere di interpunzione

`islower(c)` true se `c` è una lettera minuscola

`isupper(c)` true se `c` è una lettera maiuscola

`isspace(c)` true se `c` è un whitespace:
(spazio, tab, vertical tab, return, newline o formfeed)

`tolower(c)` ritorna il carattere minuscolo corrispondente

`toupper(c)` ritorna il carattere maiuscolo corrispondente

Accesso ai Singoli Caratteri

A volte è necessario accedere soltanto ad alcuni caratteri della stringa (per esempio trasformare la prima lettera in maiuscola)

L'operatore `[]` prende un `string::size_type` come indice dalla posizione del carattere e ritorna una reference al carattere indicato
NB indici da 0 (primo carattere) a `string::size() - 1` (ultimo carattere)

Reference al carattere → R/W permission

```
if (!s.empty()) {  
    std::cout << s << std::endl;  
    std::cout << s[0] << std::endl;  
    s[0] = std::toupper(s[0]);  
    std::cout << s << std::endl;  
}
```

```
hello World!  
h
```

```
Hello World!
```

Operazioni sulle Stringhe

<code>os << s</code>	Invia <code>s</code> sulla stream di output <code>os</code> . Ritorna <code>os</code> .
<code>is << s</code>	Legge stringhe separati da caratteri "whitespace" da <code>is</code> e li immagazzina in <code>s</code> . Ritorna <code>is</code> .
<code>getline(is, s)</code>	Legge una linea completa di input da <code>is</code> e la immagazzina in <code>s</code> . Ritorna <code>is</code> .
<code>s.empty()</code>	Ritorna <code>true</code> se <code>s</code> è vuota, <code>false</code> altrimenti.
<code>s.size()</code>	Ritorna il numero di caratteri contenuti in <code>s</code> .
<code>s[k]</code>	Ritorna una reference al carattere in posizione <code>k</code> .
<code>s1 + s2</code>	Ritorna una stringa che è il risultato della concatenazione di <code>s1</code> e <code>s2</code> .
<code>s1 = s2</code>	Rimpiazza <code>s1</code> con una copia di <code>s2</code> .
<code>s1 == s2</code>	Ritorna <code>true</code> se le stringhe sono uguali.
<code>s1 != s2</code>	Ritorna <code>true</code> se le stringhe sono diverse.

Array, Stringhe e Vettori

Lezione 8:

Parte 3. Vettori

La Classe Template `vector`

- ✓ è una raccolta di oggetti tutti dello stesso tipo
- ✓ ogni elemento della collezione ha un indice che ci permette di accedere all'oggetto corrispondente
- ✓ Per creare/manipolare oggetti della classe `vector` è necessario scrivere

```
#include <vector>
using std::vector;
```
- ✓ Il tipo `vector` è una classe template
 - un tipo `template` non è una classe nè una funzione, ma può essere pensato come una istruzione per il compilatore per generare classi o funzioni
- ✓ quando si crea un `vector` bisogna specificare il tipo degli oggetti da immagazzinare

```
vector<int> vi;           // Vettori di tipo int
vector<string> vs;      // Vettori di stringhe

vector<vector<double>> vvs; // Vettori di vettori
                        // di double
```

Creazione di un vector

- ✓ solitamente verrà creato un **vettore vuoto** del tipo richiesto:

```
vector<string> svec;    // default init: svec is empty
```

- ✓ gli **elementi del vettore** verranno aggiunti durante l'esecuzione del programma (**run-time**)
- ✓ è possibile fornire valori iniziali per un vettore
- ✓ quando si **copia un vettore** in un altro, devono essere dello **stesso tipo** e avere la **stessa dimensione**

```
vector<int> ivec;    // default init: ivec is empty
...           // inseriamo valori in ivec
vector<int> iv2(ivec);    // copia elementi da ivec a iv2
vector<int> iv3 = ivec;    // copia elementi da ivec a iv3
vector<string> svec(ivec2);    // Error: different types
```

List Initialization di un vector

- ✓ è possibile, con il nuovo standard, fornire una lista di inizializzazione per un vector

```
vector<string> svec = {"Una", "lista", "di", "stringhe"};
```

- ✓ il vettore risultante avrà 4 elementi: il primo conterrà la stringa "Una" e l'ultimo la stringa "stringhe"
- ✓ come avevamo visto, il C++ fornisce svariate forme di inizializzazione, ma non tutte possono essere utilizzate in maniera intercambiabile
- ✓ per i vettori non è possibile fornire una lista di inizializzatori usando le parentesi

```
//list initialization error  
vector<string> v1("una", "lista", "di", "stringhe");
```

Inizializzazione di vector

- ✓ è possibile creare un vettore con un numero definito di elementi e specificare il loro valore iniziale

```
vector<int> vi(10, -1); // ten int elements, each
                       // initialized to -1
```

- ✓ il valore iniziale può essere omesso, fornendo soltanto la dimensione del vector

- ✓ unica **restrizione importante**: il vettore conterrà degli **oggetti che devono possedere una inizializzazione di default**

```
vector<int> vi(10); // ten int elements, init to 0
```

```
vector<string> vs(50); // fifty string elements
                      // init to empty string
```

Riassumendo

✓ I modi per **inizializzare un vector** sono i seguenti:

```
vector<T> v1;           // Vettori di tipo T
                       // default-constructor, v1 e' vuoto

vector<T> v2(v1);      // copia in v2 tutti gli elementi di v1
                       // copy-constructor

vector<T> v2 = v1;     // equivalente a v2(v1)
                       // copia tutti gli elementi di v1 in v2

vector<T> v3(n, val);  // v3 ha n elementi con valore val

vector<T> v4(n);       // v4 ha n elementi init di default

vector<T> v5{a,b,c,...}; // v5 ha tanti elementi quanti sono
                       // gli inizializzatori

vector<T> v6 = {a,b,c,...}; // Inizializzazione equivalente
                       // a quella di v5
```

Il Metodo `push_back()`

- ✓ generalmente, quando viene creato un `vector`, non si conosce quanti elementi si andranno ad inserire
- ✓ si crea pertanto un `vettore vuoto` e si `aggiungono elementi run-time`, utilizzando il `metodo push_back()` della classe templata `vector`

```
vector<int> v; // empty vector
for (int i=0; i<100; i++) {
    v.push_back(i);
}
```

- ✓ alla fine del ciclo il `vettore conterrà 100 elementi` : gli interi da 0 a 99
- ✓ Altro esempio: `lettura di input con salvataggio del testo in un vettore di stringhe`

```
string word;
vector<string> vs; // empty vector of strings
while (cin >> word) {
    vs.push_back(word);
}
```

range for

- ✓ oltre all'operazione `push_back()`, i `vector` forniscono altre operazioni, che sono molto simili a quelle sulle stringhe
- ✓ è possibile accedere ai singoli elementi del `vector`, come accediamo ai caratteri delle stringhe

```
vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9};  
for (auto &i : v) {  
    i *= i; // prendiamo il quadrato  
}
```

- ✓ nel primo ciclo `for`, la variabile `i` è una referenza : possiamo modificare il valore immagazzinato nel vettore

```
for (auto i : v) {  
    cout << i << " ";  
}
```

- ✓ nel secondo ciclo `for`, la variabile `i` è un intero : accediamo in lettura ai valori del vettore e li stampiamo

Accesso con gli Indici

- ✓ vogliamo creare un **istogramma delle frequenze** dei risultati dei compiti di esame si Sperimentazioni 1
- ✓ dati i voti compresi tra 1 e 30 si vogliono creare **10 classi** all'interno delle quali contare i risultati

```
#include <iostream>
#include <vector>

int main(){
    vector<unsigned> histo(10, 0);
    unsigned voto;
    while (cin >> voto) {
        if (voto <=30)
            ++histo[(voto-1)/3];
    }
    // stampo i risultati
    for (auto v : histo)
        cout << v << " ";
    cout << endl;
}
```

```
$ g++ -o vec_01 vec_01.cxx
$
$ ./vec_01
29 27 24 25 26 29 30 24 22 23 18
29 28 26 26 26 30

0 0 0 0 0 1 0 4 6 6
$
```

Operazioni sui `vector`

<code>v.empty()</code>	ritorna <code>true</code> se <code>v</code> è vuoto, <code>false</code> altrimenti
<code>v.size()</code>	ritorna il numero di elementi nel vettore
<code>v.push_back(t)</code>	aggiunge l'elemento <code>t</code> alla fine del vettore
<code>v[k]</code>	ritorna una reference al vettore in posizione <code>k</code> .
<code>v1 = v2</code>	rimpiazza gli elementi di <code>v1</code> con una copia di quelli di <code>v2</code> .
<code>v1 = {a,b,c,...}</code>	rimpiazza <code>v1</code> con una copia degli elementi della lista
<code>v1 == v2</code>	ritorna <code>true</code> se hanno lo stesso numero di elementi ed ogni elemento di <code>v2</code> è in <code>v1</code>

Iteratori

- ✓ Per **navigare all'interno di un vector** (e in generale di un container in C++), è possibile usare degli **oggetti chiamati iteratori**
- ✓ Al contario dei puntatori, **non utilizziamo l'operatore address-of, (&)** per ottenere l'indirizzo dell'elemento del vettore
- ✓ Esistono **due funzioni membro della classe** :
 - `begin()` : si riferisce al **primo elemento del contenitore**
 - `end()` : si riferisce all'**elemento successivo all'ultimo**
- Se il **contenitore è vuoto** : gli iteratori ritornati da `begin()` e `end()` sono **uguali** : sono entrambi l'elemento successivo alla fine del contenitore
- Come per i puntatori, **tramite l'operatore di de-referenziazione** possiamo **accedere all'elemento** indicato dall'iteratore

```
string s("esempio di stringa");  
if (s.begin() != s.end()) { // string not empty  
    auto it = s.begin();  
    *it = toupper(*it);  
    std::cout << s << std::endl;  
}
```

esempio di stringa

Esempio di stringa

Operazioni sugli Iteratori

<code>*it</code>	ritorna una referenza all'elemento indicato dall'iteratore <code>it</code>
<code>it->mem</code>	dereferenza <code>it</code> e ritorna il membro <code>mem</code> dell'elemento indicato. Equivalente a <code>(*it).mem</code>
<code>++it</code>	incrementa di uno l'iteratore per riferirsi all'elemento successivo
<code>--it</code>	decrementa di uno l'iteratore per riferirsi all'elemento precedente
<code>it1 == it2</code>	ritorna <code>true</code> se puntano allo stesso elemento
<code>it1 != it2</code>	ritorna <code>true</code> se si riferiscono ad elementi diversi

Aritmetica degli Iteratori

<code>*iter</code>	ritorna una referenza all'elemento indicato dall'iteratore
<code>iter + n</code>	scorre il contenitore di n posizioni in avanti
<code>iter - n</code>	scorre il contenitore di n posizioni all'indietro
<code>iter += n</code>	assegnazione composta per la somma di iteratori
<code>iter -= n</code>	assegnazione composta per la differenza di iteratori
<code>iter1 - iter2</code>	la sottrazione di due iteratori fornisce un numero che sommato all'iteratore di destra fornisce l'iteratore di sinistra
<code>>, >=, <, <=</code>	operatori relazionali su iteratori