



# Laurea triennale in Informatica

*modulo (CFU 6) di*

## Programmazione II e Lab.

# prof. Mariarosaria Rizzardi

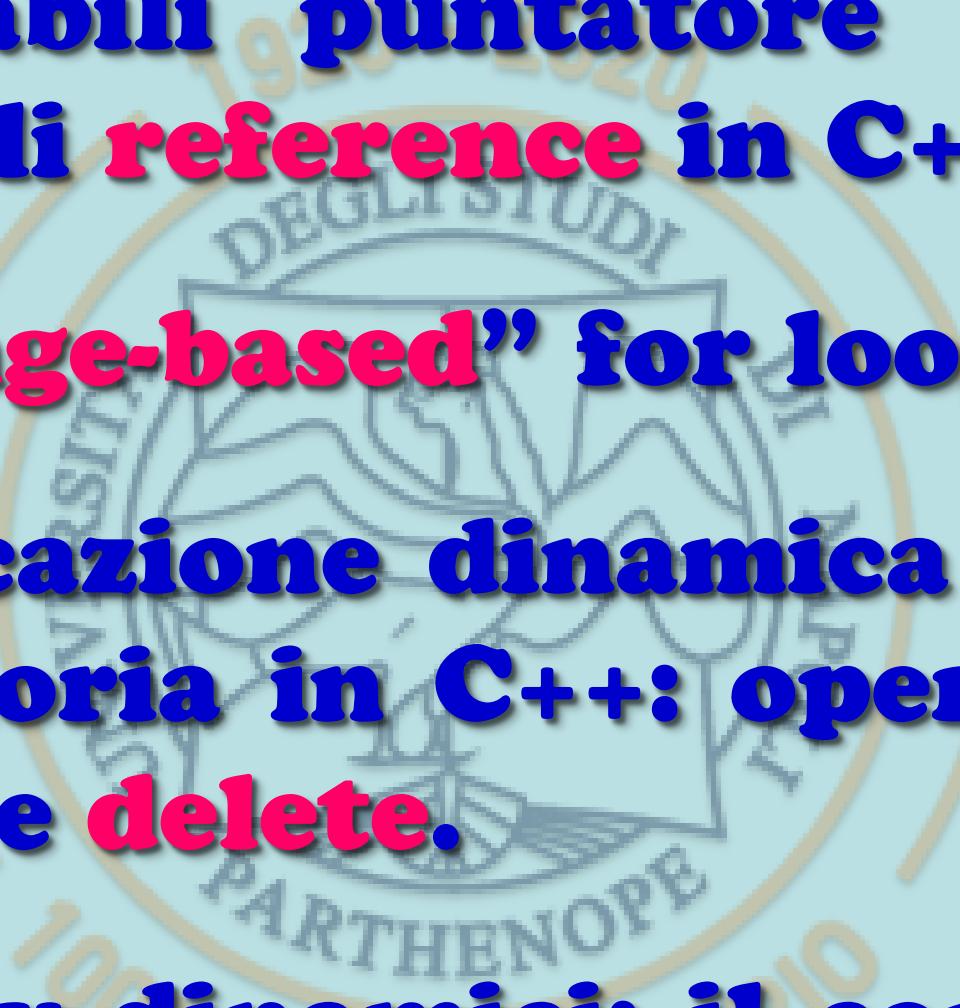
Centro Direzionale di Napoli – Isola C4

stanza: n. 423 – IV piano Lato Nord

tel.: 081 547 6545

email: mariarosaria.rizzardi@unipARTHENOPE.it

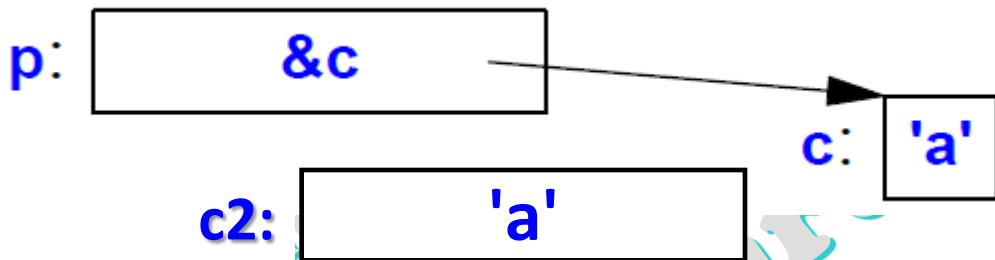
- **Variabili puntatore e variabili reference in C++.**
- **“Range-based” for loop.**
- **Allocazione dinamica della memoria in C++: operatori new e delete.**
- **Array dinamici: il contenitore sequenziale vector.**



# Richiami sui puntatori

```
char c = 'a';
char* p = &c;
char c2 = *p;
```

operazione di dereferenziazione



```
int* p1;
char** p2;
float* p3[5];
int (*fp)(char*);
int* f(char*);
```

Puntatore a int  
 Puntatore a char\*  
 Array di 5 float\*  
 Puntatore a funzione con parametro char\* e valore di ritorno int  
 Funzione con parametro char\* e valore di ritorno int\*

variabili già definite

```
p1++;
p2++;
p3[0]++;
```

Incrementa di ...?... byte l'indirizzo contenuto in p1  
 Incrementa di ...?... byte l'indirizzo contenuto in p2  
 Incrementa di ...?... byte l'indirizzo contenuto in p3

In **C** **NULL** è la macro per indicare il puntatore nullo (un puntatore che non punta a niente). In **C++ nullptr** è la costante per indicare il puntatore nullo.

# Richiami sui puntatori

```
int* p1;  
char** p2;  
float* p3[5];
```

variabili già definite

```
p1++;  
p2++;  
p3[0]++;
```

Puntatore a int

Puntatore a char\*

Array di 5 float\*

Incrementa di 4 byte l'indirizzo contenuto in p1

Incrementa di 8 byte l'indirizzo contenuto in p2

Incrementa di 0 byte l'indirizzo contenuto in p3

Perché?

# Variabili reference

Il C++, come il C, prevede la dichiarazione di:

- variabili mediante nome:      **int v;**
- variabili puntatore:      **int\* pt; pt=&v;**

In aggiunta il C++ prevede anche la dichiarazione di **variabili reference**:

**int& r=v;**

da questo momento in poi **r** e **v** rappresentano lo stesso valore. L'istruzione **r++** equivale a **v++**, mentre **pt++** incrementa di 8 byte l'indirizzo di memoria.

Un **reference**, quando dichiarato, deve puntare ad una variabile già dichiarata; quindi la dichiarazione ne prevede anche l'inizializzazione.

L'indirizzo cui punta una variabile **reference** non può essere cambiato.

## Restrizioni:

- Non può esserci un **reference** a una variabile **reference**.
- Non si può creare un array di **reference**.
- Non si può creare un puntatore a un **reference**, cioè l'operatore & (indirizzo) non è applicabile ad un **reference**.
- I **reference** non sono consentiti per i campi di bit (in una struct).

# Esempi d'uso dei reference

## passaggio dei parametri "per reference" in C++

```
#include <iostream>
void swap(int&, int&);
using namespace std;
int main()
{
    int a=2, b=3;
    cout << "prima dello scambio: a=" << a << ", b=" << b << endl;
    swap(a,b);
    cout << "dopo lo scambio:      a=" << a << ", b=" << b << endl;
    return 0;
}
```

```
void swap(int &p, int &q)
{// scambio del contenuto di p e q
    int tmp=p;
    p=q;
    q=tmp;
}
```

## "range-based" for loop in C++ (C++11)

```
#include <iostream>
using namespace std;
int main()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};
    for (auto &x : v)
        cout << "++x = " << ++x << endl;
    return 0;
}
```

# Ancora sul range-based for loop

```
#include <iostream>
using namespace std;
int main()
{ int v[] = {0,1,2,3,4,5,6,7,8,9};
  for ...
  return 0; }
```

```
for (auto &x : v)
  cout << "++x = " << ++x << endl;
```

Per lavorare con le voci originarie dell'array potendole anche modificare.

```
++x = 1
++x = 2
++x = 3
++x = 4
++x = 5
++x = 6
++x = 7
++x = 8
++x = 9
++x = 10
```

```
v[0] = 1
v[1] = 2
v[2] = 3
v[3] = 4
v[4] = 5
v[5] = 6
v[6] = 7
v[7] = 8
v[8] = 9
v[9] = 10
```

vettore  
modificato

```
++x = 0
++x = 1
++x = 2
++x = 3
++x = 4
++x = 5
++x = 6
++x = 7
++x = 8
++x = 9
```

```
v[0] = 0
v[1] = 1
v[2] = 2
v[3] = 3
v[4] = 4
v[5] = 5
v[6] = 6
v[7] = 7
v[8] = 8
v[9] = 9
```

vettore non  
modificato

```
for (auto x : v)
  cout << "++x = " << ++x << endl;
```

Per lavorare con una copia delle voci originarie dell'array non potendole modificare.

```
for (auto const &x : v)
  cout << "x = " << x << endl;
// cout << "++x = " << ++x << endl;
//
errore: read-only variable
```

Per lavorare con le voci originarie dell'array senza poterle modificare.

```
++x = 1
++x = 2
++x = 3
++x = 4
++x = 5
++x = 6
++x = 7
++x = 8
++x = 9
++x = 10
```

```
v[0] = 0
v[1] = 1
v[2] = 2
v[3] = 3
v[4] = 4
v[5] = 5
v[6] = 6
v[7] = 7
v[8] = 8
v[9] = 9
```

# Allocazione dinamica in C++

L'allocazione dinamica in C++ è gestita mediante gli operatori **new** e **delete**:

- **new** per allocare l'area dinamica (nella memoria heap);
- **delete** per la sua deallocazione.

Gli operatori **new** e **delete** sono come le funzioni **malloc()** e **free()** del C. Non esiste in C++ qualcosa come **realloc()**. L'alternativa è usare la classe **vector**.

## Esempi

```
int* pt = new int;
cout << "*pt = " << *pt << endl;
*pt = 7;
int* pt = new int(7);
cout << "*pt = " << *pt << endl;
delete pt;
```

alloca dinamicamente un'area per un intero **int** (puntata da **pt**) e vi assegna il valore 7

```
int* pt = new int[100];
if (pt == nullptr)
    // errore...
...
delete [] pt; dealloca l'array
```

alloca dinamicamente un'area per 100 interi (puntata da **pt**). Se l'allocazione non avviene, **new** restituisce **nullptr**.

# Esempio 1: dangling pointer

Un **dangling pointer** è un puntatore (non nullo) che punta a dati non più validi, cioè punta ad un'area di memoria deallocated.

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

```
int main()
{
    int* pt = new int [5];
    *pt=7;

    cout << "dopo new:      pt = " << pt << "\t *pt = " << *pt << endl;
    cout << "dopo delete: pt = " << pt << "\t *pt = " << *pt << endl;

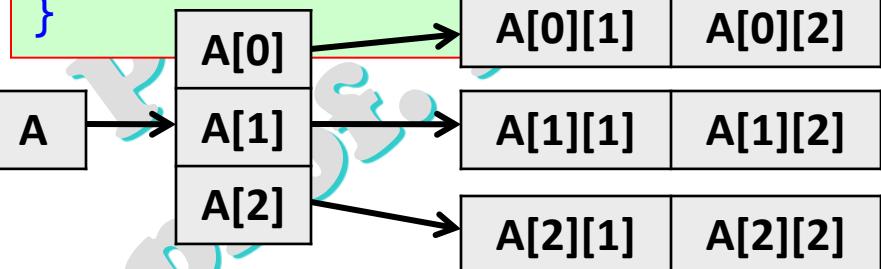
    delete [] pt;
    return 0;
}
```



## Esempio 2: matrici mediante new e delete

Una **matrice A(m×n)** è allocata dinamicamente come array di puntatori ad array

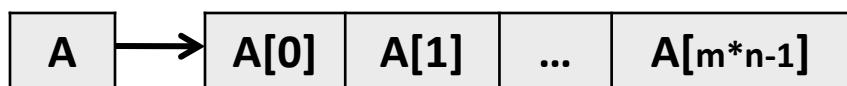
```
#include <iostream>
using namespace std;
int main()
{   int m=3, n=2;      alloca
    int** A = new int* [m];
    for (int i=0; i<m; i++)
        A[i] = new int [n];
...
for(int i=0; i<m; i++)
    delete [] A[i];
delete [] A;
dealloca
return 0;
}
```



Si usano i doppi indici come se fosse statica

Una **matrice A(m×n)** è allocata dinamicamente come un vettore. Il programmatore si fa carico di esplicitare l'indice per l'accesso alle componenti.

```
#include <iostream>
using namespace std;
int main()
{   int m=3, n=2;      alloca
    int* A = new int [m*n];
    // elem i,j: A[i*n+j] per righe
    //           : A[j*m+i] per colonne
...
delete [] A;
dealloca
return 0;
}
```



# La classe **vector**

I **vector** sono contenitori sequenziali per rappresentare array che possono **cambiare size** durante l'esecuzione, a differenza degli array allocati dinamicamente con **new**.

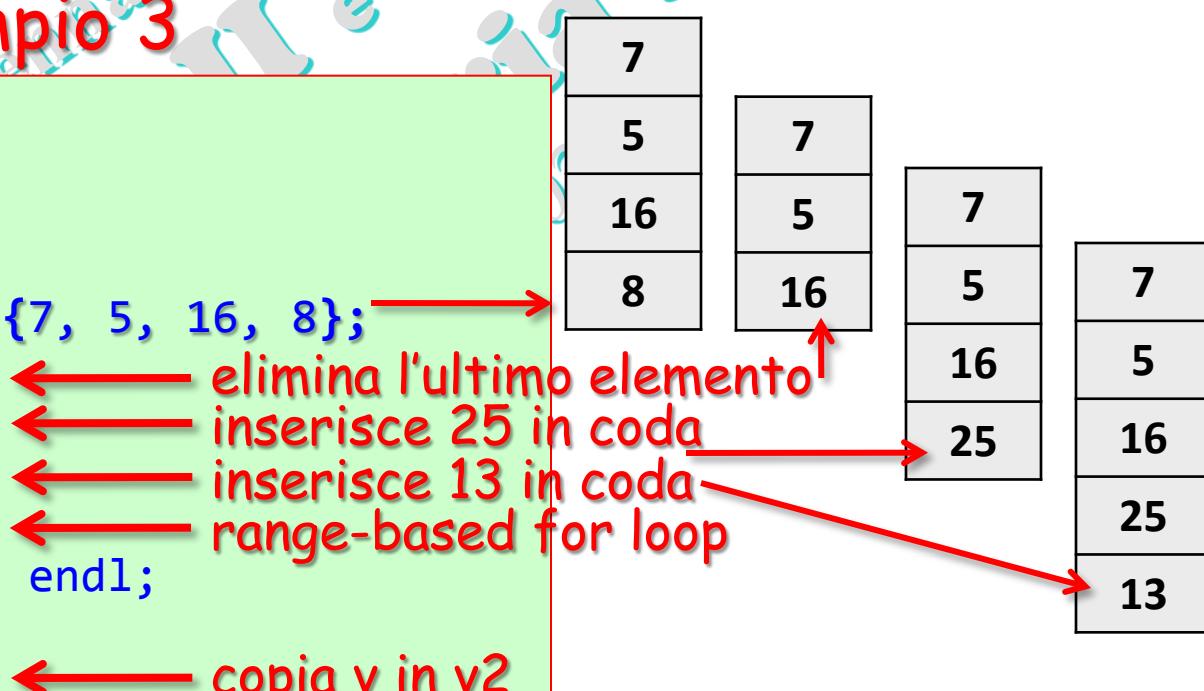
## Operazioni sui vector:

Accedere a qualunque elemento.

Aggiungere o eliminare elementi dovunque.

## Esempio 3

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v = {7, 5, 16, 8};
    v.pop_back();           ← elimina l'ultimo elemento
    v.push_back(25);       ← inserisce 25 in coda
    v.push_back(13);       ← inserisce 13 in coda
    for(auto &n : v)      ← range-based for loop
        cout << n << endl;
    vector <int> v2;
    v2=v;                  ← copia v in v2
    return 0;
}
```



# Esempio 4a: matrici mediante vector

Una matrice A( $m \times n$ ) allocata dinamicamente come vector di vector:

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

using namespace std;

int main()
{
    int m=3, n=2;
    vector<vector<int>> A {{1,2,3},{4,5},{6}};

    cout << "A.size() = " << A.size() << endl;
    cout << "A[0].size() = " << A[0].size() << endl;
    cout << "A[1].size() = " << A[1].size() << endl;
    cout << "A[2].size() = " << A[2].size() << endl;
    for (int i=0; i<m; i++)
    {
        for (int j=0; j<A[i].size(); j++)
            cout << A[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

... // mediante enumerazione di elementi  
**vector<vector<int>> A {**

1	2
3	4
5	6

...

{1,2},  
{3,4},  
{5,6}};

A.size() = 3  
A[0].size() = 3  
A[1].size() = 2  
A[2].size() = 1

righe di size diverso

1	2	3
4	5	
6		

## Esempio 4b: matrici mediante vector

Una matrice A( $m \times n$ ) allocata dinamicamente come vector di vector:

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

using namespace std;

int main()
{
    int m=3, n=2;
    vector<vector<int>> A (m);
    cout << "A.size() = " << A.size() << endl;

    for(int i=0; i<m; i++)
        A[i].resize(n);

    cout << "A[2].size() = " << A[2].size() << endl;

    for (int i=0; i<m; i++)
    {
        for (int j=0; j<n; j++)
            cout << A[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

0	0
0	0
0	0

inizializza a 0

## Esempio 4c: matrici mediante vector

Una matrice  $A(m \times n)$  allocata dinamicamente come vector di vector:

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

using namespace std;

int main()
{   int m=3, n=2;
    vector<vector<int>> A;
    A.resize(m, vector<int>(n));
    for (int i=0; i<m; i++)
    {
        for (int j=0; j<n; j++)
            cout << A[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

0	0
0	0
0	0

Riccardi

Pro

# Esempio 4d: matrici mediante vector

Una matrice A( $m \times n$ ) allocata dinamicamente come vector di vector:

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

using namespace std;

int main()
{
    int m=3, n=2;
    vector<vector<int>> A (m, vector<int>(n,5));

    for (int i=0; i<m; i++)
    {
        for (int j=0; j<n; j++)
            cout << A[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

...

**vector<vector<int>> A (m, vector<int>(n,5));**

5 5  
5 5  
5 5

**fill constructor**  
per inizializzare.

0 0  
0 0  
0 0

## Esempio 5: vector di stringhe

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector <string> colori;
    colori.push_back("Blue");
    colori.push_back("Red");
    colori.push_back("Orange");
    colori.push_back("Yellow");
    int k=0;
    for (auto &c : colori) // range-based for loop
        cout << "in [" << (k++) << "]: " << c << endl;
    cout<<endl;
    cout << "primo elem.: " << colori.front() << endl;
    cout << "ultimo elem.: " << colori.back() << endl;
    colori.clear(); // svuota l'array, lasciando il contenitore vuoto
    return 0;
}
```

Non si può usare il *range-based for loop* con gli array allocati dinamicamente (*new* e *delete*), perché il compilatore non ne conosce l'inizio e la fine. Si può fare con i *vector*.

```
in [0]: Blue
in [1]: Red
in [2]: Orange
in [3]: Yellow

primo elem.: Blue
ultimo elem.: Yellow
```

# Esempio 5b: find in vector di stringhe

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm> // per find
using namespace std;
int main()
{
    vector<string> colori;
    colori.push_back("Blue");
    colori.push_back("Red");
    colori.push_back("Orange");
    colori.push_back("Yellow");

    string str("Orange"); // stringa da cercare
    vector<string>::iterator it=find(colori.begin(),colori.end(),str);
    if (it != colori.end())
    {
        cout << "stringa \" " << str << "\" trovata in pos.: "
            << distance(colori.begin(),it)+1 << endl;
        cout << "*it = \" " << *it << "\" " << endl;
    }
    return 0;
}
```

**iteratore**: astrae il concetto di puntatore a un elemento del **contenitore**.  
Meccanismo che rende possibile disaccoppiare gli algoritmi dai contenitori.

↓                            ↓  
                iteratori  
1° elemento              oltre l'ultimo elem.

calcola il numero di elementi tra i due iteratori

stringa "Orange" trovata in pos.: 3  
 \*it = "Orange"

# La classe template vector

<http://www.cplusplus.com/reference/vector/vector/>

The screenshot shows the C++ Reference documentation for the `vector` class. The left pane lists the public member functions, and the right pane lists the non-member overloads.

**member functions:**

- `vector::vector`
- `vector::~vector`
- **`vector::assign`**
- `vector::at`
- `vector::back`
- `vector::begin`
- `vector::capacity`
- `vector::cbegin`
- `vector::cend`
- `vector::clear`
- `vector::crbegin`
- `vector::crend`
- `vector::data`
- `vector::emplace`
- `vector::emplace_back`
- `vector::empty`
- `vector::end`
- `vector::erase`
- `vector::front`

**non-member overloads:**

- `relational operators (vector)`
- `swap (vector)`

# Iteratori

```
...  
vector<int> vec;  
vec.push_back(1);  
vec.push_back(4);  
vec.push_back(8);  
...
```



**vec.begin()**

punta alla prima componente

**vec.end()**

punta dopo l'ultima componente

## Vecchio approccio (C-style)

```
...  
for (int k=0; k<vec.size(); k++)  
    cout << vec[k] << " ";
```

## Nuovo approccio (C++ STL)

```
...  
vector<int>::iterator It;  
for (It = vec.begin(); It < vec.end(); It++)  
    cout << *It << " ";
```