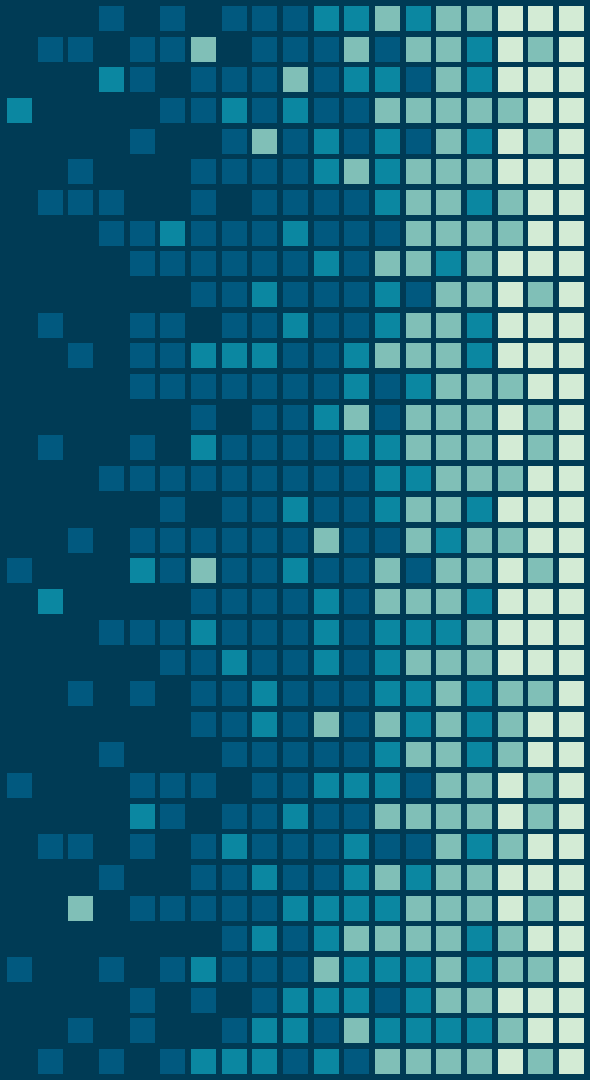
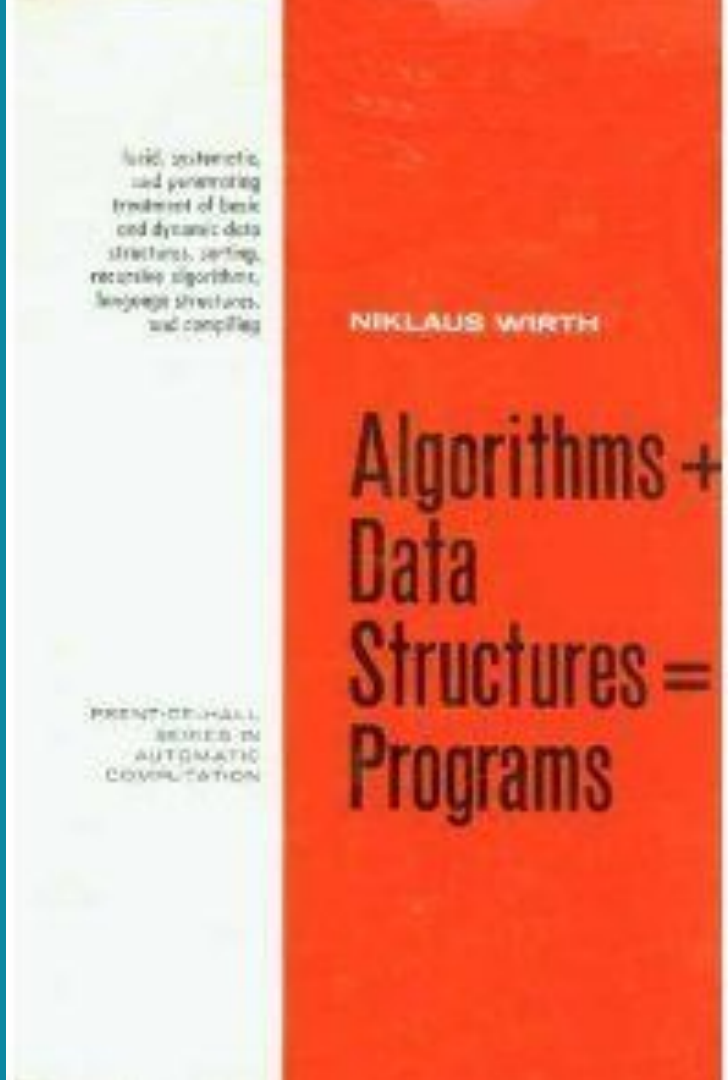
C++ from zero to hero
Parte 2



> *Software development is technical activity conducted by human beings.*

Niklaus Wirth



1.
Strutture dati

Una prima introduzione

Variabili semplici

Finora abbiamo considerato variabili semplici (detti anche scalari o primitivi). In sostanza, si tratta di dati che sono in grado di contenere UN SOLO VALORE.

Tutti i linguaggi di programmazione permettono di superare questa limitazione in diversi modi.

Variabili strutturate

Le variabili di questo tipo sono dette strutturate, o composte. Possiamo distinguere due grandi categorie: una strutturazione

- Per molteplicità
- Per tipo

Per molteplicità

Si riferisce alle variabili in grado di memorizzare un certo numero di variabili dello stesso tipo.

Il prototipo di questo tipo di variabili sono gli array, presenti in tutti i linguaggi, ma esistono tanti tipi di strutture dati che differiscono per tipologia, metodo di accesso e prestazioni.

Per tipo

Si riferisce alle variabili in grado di memorizzare un certo numero (solitamente fisso) di variabili di tipo diverso

Il prototipo di questo tipo di variabili sono gli struct, e la definizione di tipi di dati, che si sono quindi evoluti nella paradigma a oggetti.

In realtà....

Il confine è labile, dato che i due aspetti possono essere combinati (array di struct, struct di array) secondo il “principio Matrioska”.

Ma per fare un po’ di ordine manterremo questa distinzione.

2.
Array e i suoi cugini

Array classici

Formalmente, un array è una sequenza di elementi dello stesso tipo, immagazzinata in locazioni di memoria contigue. La definizione tradisce la sua natura di basso livello. Caratteristiche

- Struttura statica, inizializzata in compile time ~~o in runtime~~.
- Si accede a ogni elemento tramite indice.
- Gli indici sono 0-based, senza alcun tipo di controllo
- Sono strutture “stupide”, identificate semplicemente dall’indirizzo base e dal tipo di dato.

Dichiarazione C-style

La dimensione degli array **deve** essere un valore costante, perché il compilatore deve sapere quanto spazio riservare:

```
int pippo[100];
int pippo[8][8];
```

Dichiarazione C-style

In teoria sarebbe possibile creare gli array in fase di runtime in questo modo:

```
int dim=4;
int pippo[dim];
```

Tuttavia tale pratica è sconsigliata (prevista dal C99 ma non prevista dallo standard C++) anche se la maggior parte dei compilatori lo accettano.

Inizializzazione

Al contrario dei tipi scalari, gli array non sono in genere inizializzati dal compilatore. Dovete quindi farlo nel programma o durante la dichiarazione:

```
int danger[]; // brrr...
int pippo[3] = {10,20,30};
int pluto[] {10,20,30}; // C++11
int orazio[30]={}; // tutti a 0
```

Uso

Per accedere agli elementi si usa l'operatore `[]`.

Il tempo di accesso è costante (*random access*).

```
int arr[100]{};
for (int i=0; i<100; i++)
   {arr[i]=1;}
```

Problemi degli array

Seppur semplici e immediati, gli array sono problematici.

- Forte possibilità di errori
- Difficoltà di debug
- Ripetitività di operazioni elementari
- Dimensioni statiche (note a compile-time)

IS THERE A BETTER WAY?

3.

<array> & <vector>

...a different way: STL

Introducing the STL

La **S**tandard **T**emplate **L**ibrary (**STL**) è una libreria software per il linguaggio di programmazione C++ che ha avuto una grande influenza nella realizzazione della Standard Library, fornita assieme a ogni compilatore.

Formato dalle seguenti componenti principali:

- algorithms (algoritmi)
- containers (contenitori)
- functions (funzioni, “member functions”)
- iterators (iteratori).

Containers

Cominciamo con un punto fondamentale della STL: i containers

Un **container** è un oggetto contenitore che gestisce una collezione di altre entità, dette *elementi*.

Sono implementati tramite *template*, che permettono una grande flessibilità di utilizzo.

Si tratta quindi di oggetti, qualcosa in più rispetto alle variabili classiche, ma che spesso sono indistinguibili nell'uso (*syntactic sugar magic*)

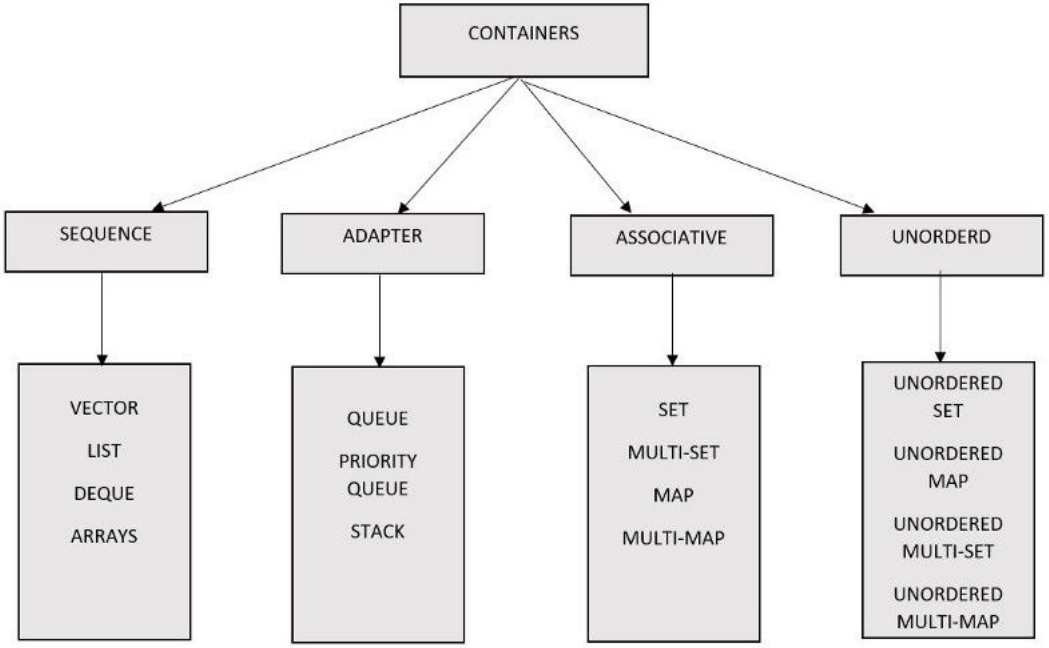
I container, in sintesi

Vantaggi:

- Riducono le possibilità di errore
- Semplificano l'uso
- Accelerano *alcune* operazioni

Esistono più di 10 container: noi ne guarderemo in dettaglio solo alcuni

Tassonomia


CONTAINERS
SEQUENCE
ADAPTER
ASSOCIATIVE
UNORDERD
VECTOR
LIST
DEQUE
ARRAYS
QUEUE
PRIORITY QUEUE
STACK
SET
MULTI-SET
MAP
MULTI-MAP
UNORDERED SET
UNORDERED MAP
UNORDERED MULTI-SET
UNORDERED MULTI-MAP

<array>

<array> è un wrapper attorno agli array C, una versione "Cipiupiuata" insomma. Fornisce alcune funzioni e memorizza la dimensione. Si consiglia di usarli SEMPRE al posto della sintassi classica, anche se può risultare aggrovigliata. Esempio:

```
array<int,4> a = {10, 20, 30, 40};

array<std::array<int,3>,2> a {{
    {{1,2,3}},{{4,5,6}} }};

array a = {10, 20, 30, 40};
```

Bonus: perché tutte queste parentesi?

L'ultima sintassi è disponibile solo nelle versioni più recenti del linguaggio.

<vector>

<vector> è un **array** espandibile che può **crescere** o **diminuire** in dimensioni. Si possono i**nserire *efficientemente* elementi** in coda, meno efficientemente **in altre posizioni**.

vector garantisce la disposizione degli elementi in modo **contiguo** (come un array). Si può quindi utilizzare per interoperare con API C (es: si usa **v.data()** per accedere al "buffer" di elementi interno).

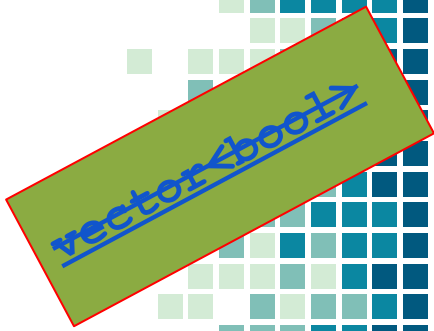
In linea di massima, nel caso unidimensionale, **il 99% dei casi conviene usare un vector** anziché un array.

Differenze principali

array	vector
Dimensione statica	Dimensione dinamica
Storage sequenziale	Storage sequenziale
Allocazione su stack	Allocazione su heap
Struttura immutabile	Struttura modificabile
Massima efficienza	Massima flessibilità

Esempio

```
#include <vector>
// vector vuoto di interi
vector<int> vi;
// vector di float con tre elementi a 0.0f
vector<float> vf(3);
// array di float non inizializzati
array<float, 3> vf;
// vector di string pre-inizializzato
vector<std::string> vs{ "pippo", "pluto"};
```



Interoperabilità con C

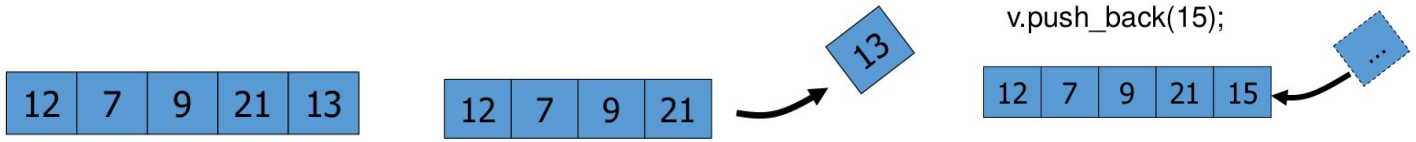
```
int arr[] = { 10, 20, 30, 40, 50};
// vector da C-array
vector<int> v(arr, arr + 3);

// oppure (meglio)
vector<int> v(begin(arr), end(arr));

// riempire un array da vector
// questa sintassi sarà chiara tra poco
std::copy_n(begin(v), size(arr), begin(arr));
```

Inserimenti e rimozioni

vector<int> v{12, 7, 9, 21, 13 }; v.pop_back(); v.push_back();



Possibile inserire in altre posizioni, ma non è così semplice e soprattutto così veloce. Usare funzione **insert()**.

Metodi principali

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

Container derivati

Si tratta essenzialmente di vector che espongono una interfaccia più semplice in modo da simulare altre strutture dati (façade?). Esempio

Stack (LIFO)

Queue (FIFO)

Priority Queue (~Heap)

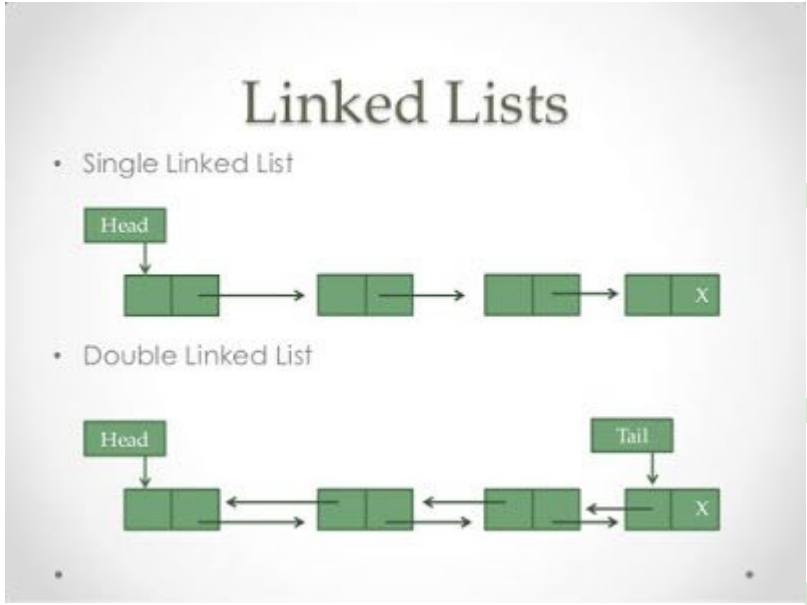
Prestazioni

Container	Insertion	Access	Erase	Find
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)
priority_queue	O(log n)	O(1)	O(log n)	-

4.
Altre strutture

Liste

Una lista è una struttura dati in cui, per ogni elemento, è definito un elemento successivo. In certi casi, è definito anche un elemento precedente.



Uso

Le liste si manipolano (quasi) esclusivamente tramite gli iteratori, che tratteremo a seguire. Per il momento un breve assaggio.

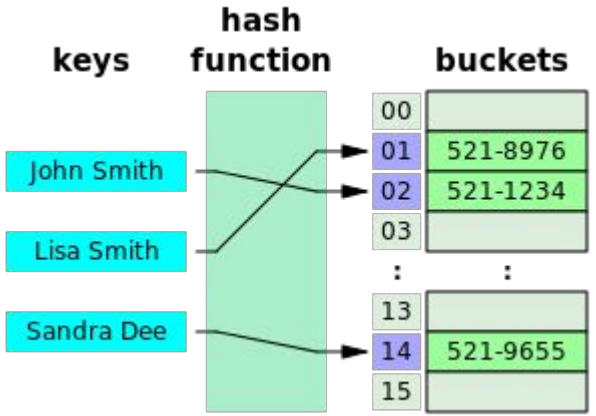
```
forward_list<int> lista{ 1, 2, 3, 4, 5 };

for (auto i : lista)
    cout << ' ' << i;
```

Mappe

Le mappe sono strutture che permettono di recuperare i dati in modo rapido. Sono basate su funzioni di hash che trasformano la chiave in modo da accedere in modo diretto all'elemento richiesto.

In STL le strutture basate sulle mappe sono i **set**, **unordered_set** (solo chiavi) e **map** e **unordered_map** (chiavi e valori)



Possibile schema di una struttura "Unordered" (basata su hash)

Differenza ordered/unordered

I container ordinati (**map**, **set**, **multimap**, **multiset**), mantengono un ordinamento tra gli elementi (solitamente sono implementati con *alberi binari di ricerca* - es: red-black perché sono *auto-bilanciati*).

Richiedono quindi un **criterio di ordinamento**, che di default è `operator<` (detto **less**). Es: 1 < 2 < 3; "marcello" < "pippo"

Due elementi sono "equivalenti" se si verifica:

!(A < B) && !(B < A)

Es: !(2 < 2) && !(2 < 2) VERO (infatti 2==2)

!(1 < 2) && !(2 < 1) FALSO (infatti 1!=2)

Esempio

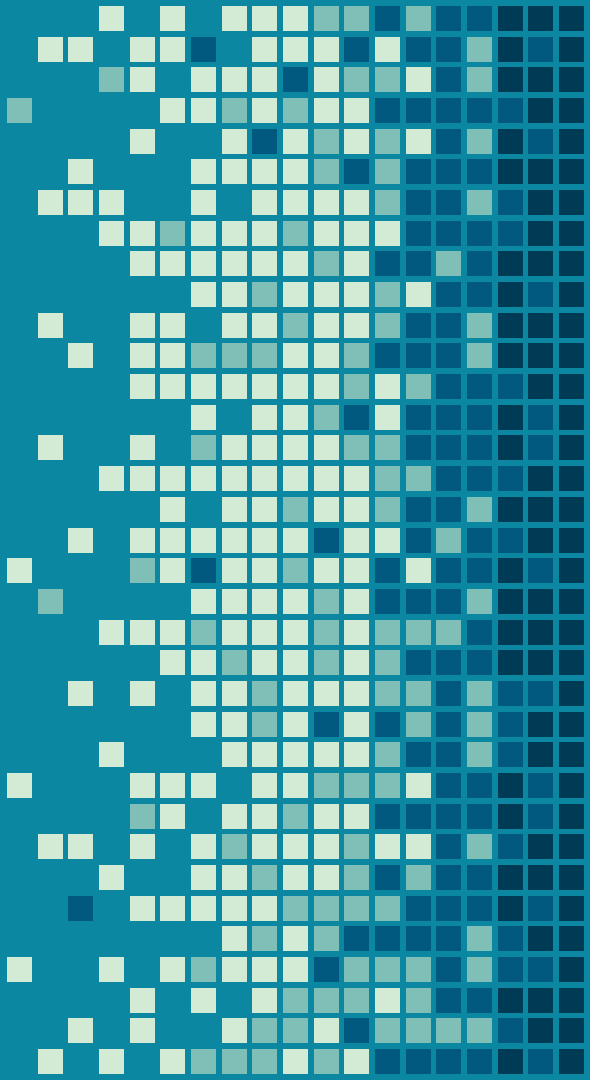
Le mappe simulano anche gli array associativi presenti in altri linguaggi

```
 std::map<std::string, int> pianeti {
    {"mercurio", 1},
    {"terra", 3}
};
pianeti["marte"] = 5; // inserisce o aggiorna
pianeti.emplace("marte", 4); // inserisce ma non aggiorna
cout << pianet.count(“marte”); // 1
```

Prestazioni

Container	Insertion	Access	Erase	Find
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)
set / map	O(log n)	-	O(log n)	O(log n)
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)

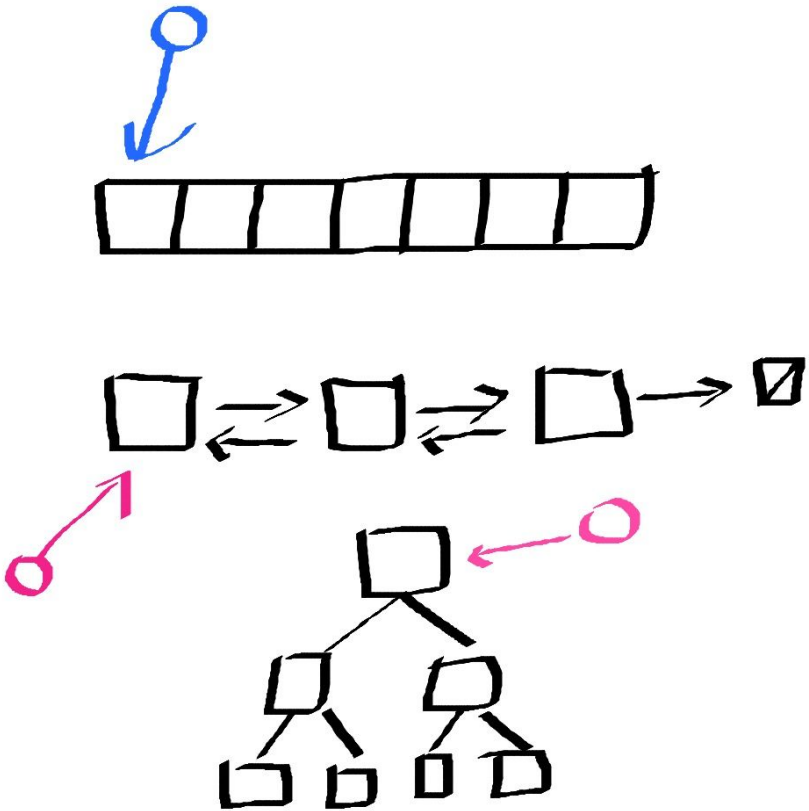
> *“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”*
>
> *Linus Torvalds.*



5.
Iteratori

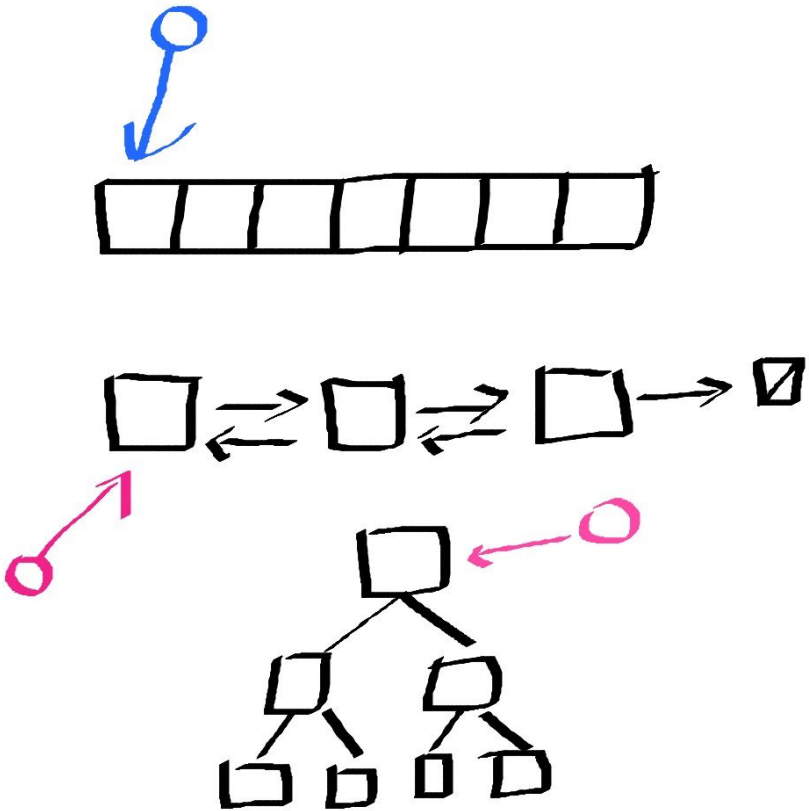
Che sono gli iteratori?

- Un iteratore è una classe che astrae la navigabilità su una struttura dati nascondendone i dettagli implementativi.
- Il suo uso quindi è simile anche se i dati sui quali lavora sono radicalmente diversi



Che sono gli iteratori?

- Sono entità simili ai puntatori, usati per accedere a un elemento di un tipo di dati.
- Tipicamente usati per scandire l'intero contenuto: tale processo si chiama, poco sorprendentemente,
- "Iterazione di un contenitore"



Filosofia di fondo

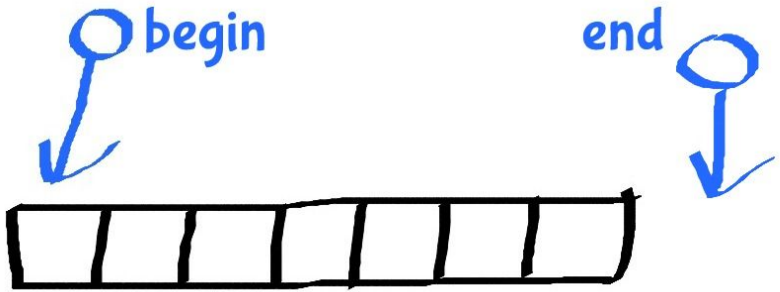
- Idealmente vorremmo che l'iteratore fosse semplice da usare come nel codice qui a destra ("javoso").
- In realtà la natura del linguaggio ci chiede di gestire alcune cose in modo più esplicito
-

In pseudo linguaggio

```
Stampa (Iteratore)
    while(Iteratore.haSuccessivo):
        Output (iteratore.valore);
        iteratore.successivo()
```

Filosofia di fondo

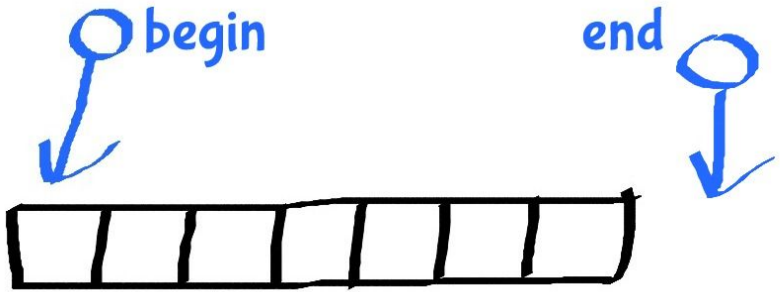
- In realtà in C++ per usare gli iteratori dobbiamo sempre definire un intervallo (tech: *range*)
- Questo intervallo è detto "semiaperto", prendendolo dalla terminologia matematica: comprende l'inizio, ma non la fine.



Filosofia di fondo

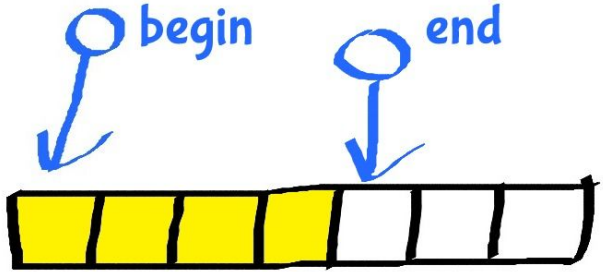
I vantaggi di questa convenzione sono principalmente due:

- Gestisce anche il caso "anomalo" di intervallo vuoto (begin, end sono coincidenti);
- la condizione di terminazione di un ciclo è banalmente **`begin != end`**.



In sintesi

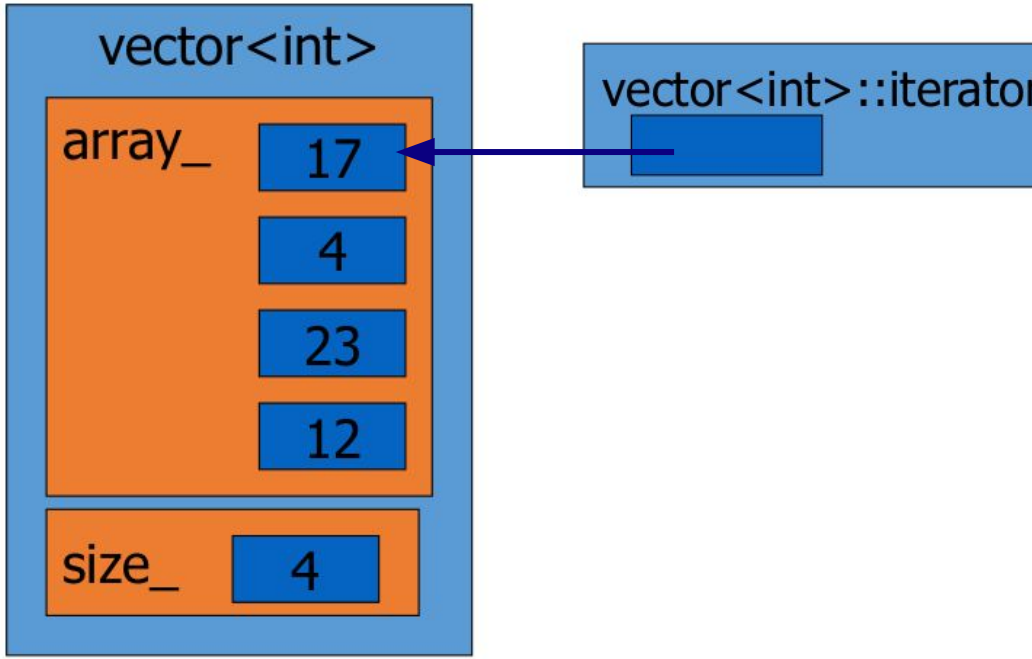
- Permettono di astrarre dal tipo di dato utilizzato
- Forniscono un metodo per accedere a un singolo elemento del container utilizzato
- Forniscono un metodo per passare all'elemento successivo (e al precedente
- Permettono di stabilire intervalli sui quali lavorare
  - Tipicamente dall'inizio alla fine (.begin, .end)
  - Possibile anche intervalli arbitrari
    It.begin() e it.begin()=+4



Iteratori in STL

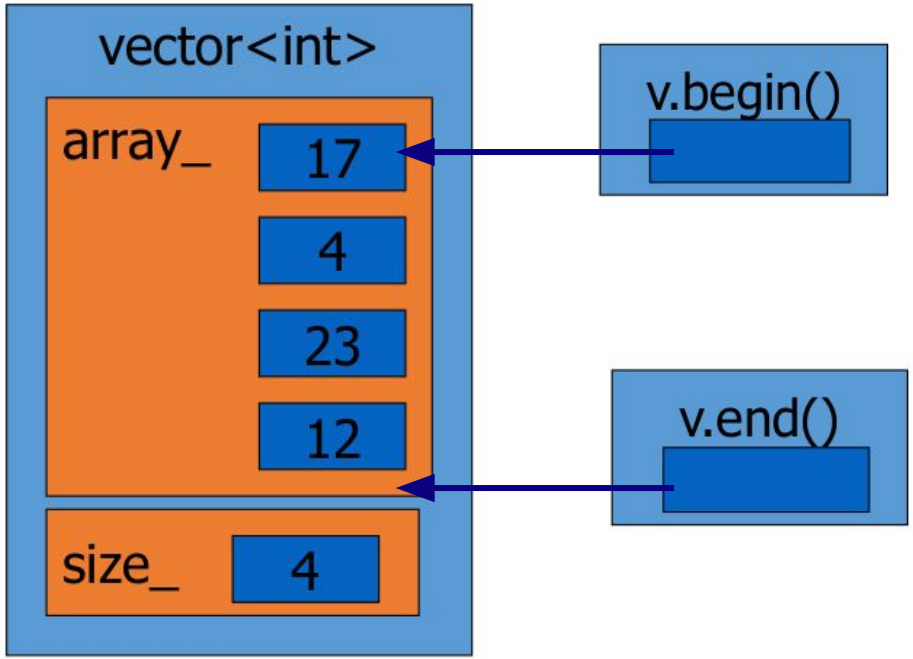
In generale , un iteratore è definito in questo modo, `vector<int>` è `vector<int>::iterator`

Tuttavia, spesso è meglio far fare il lavoro al compilatore, dichiarandolo con il tipo `auto`. Ci penserà lui!



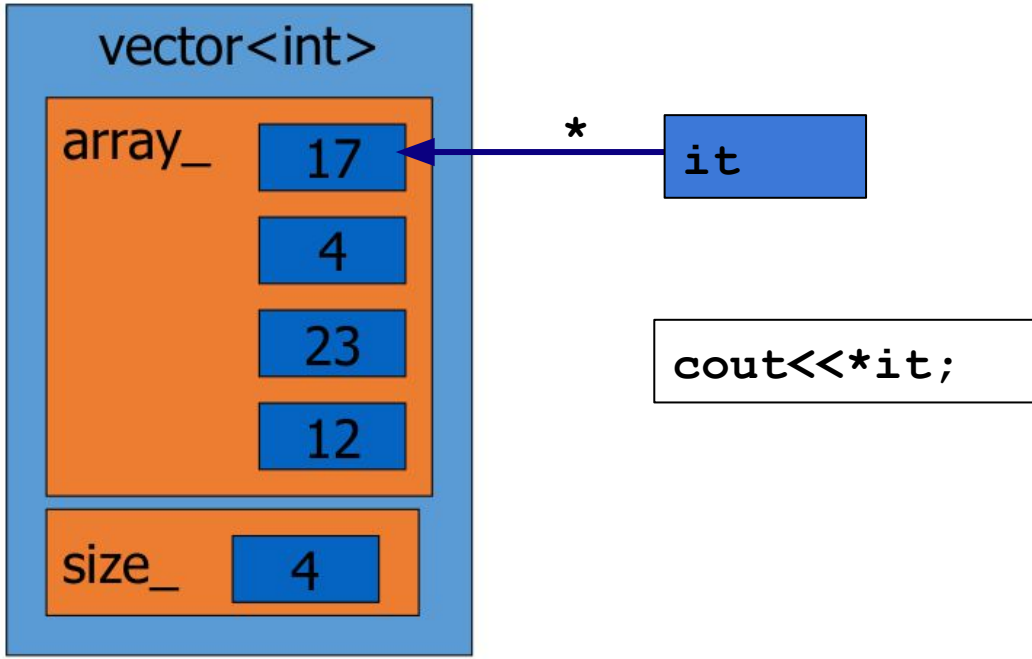
Inizio e fine

Per ottenere il range che interessa si usano i metodi e funzioni `.begin()` and `.end()`



Accesso all'elemento

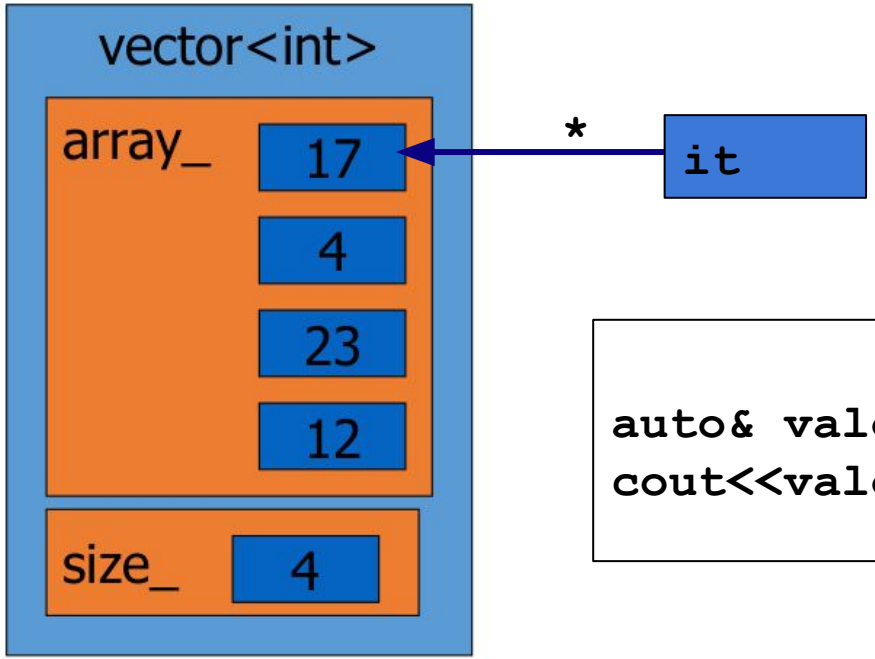
L'iteratore fornisce un **riferimento** all'elemento, per cui per accedere occorre un'operazione di deferenziazione (operatore asterisco).



Accesso all'elemento

In C++ il modo più elegante è quello di usare una variabile apposta come questa:

```
auto& valore = *it;
```

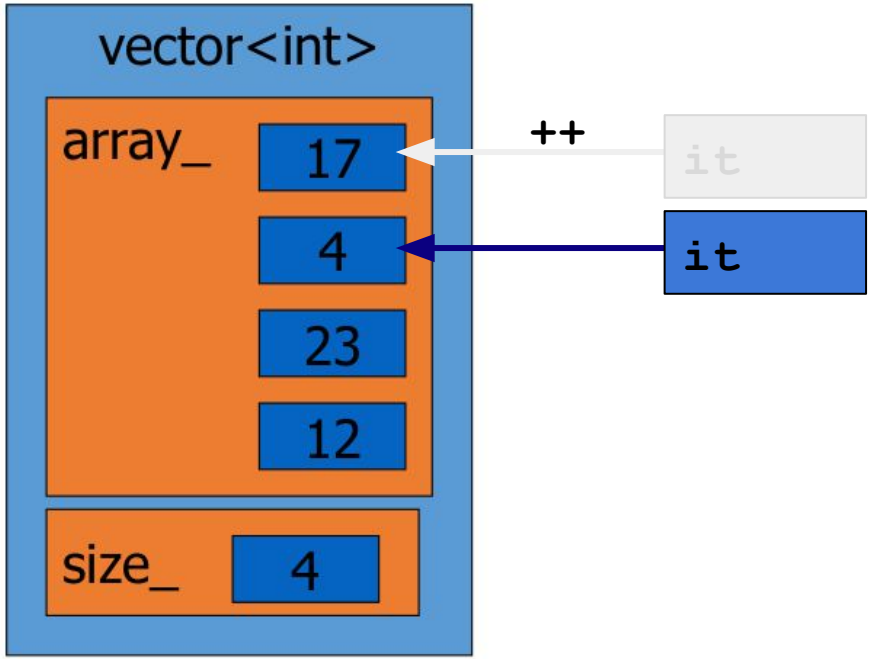


Scansione

Per passare agli elementi successivi o precedenti si usano varie funzioni come il noto operatore ++ , nella sua versione ++it o it++.

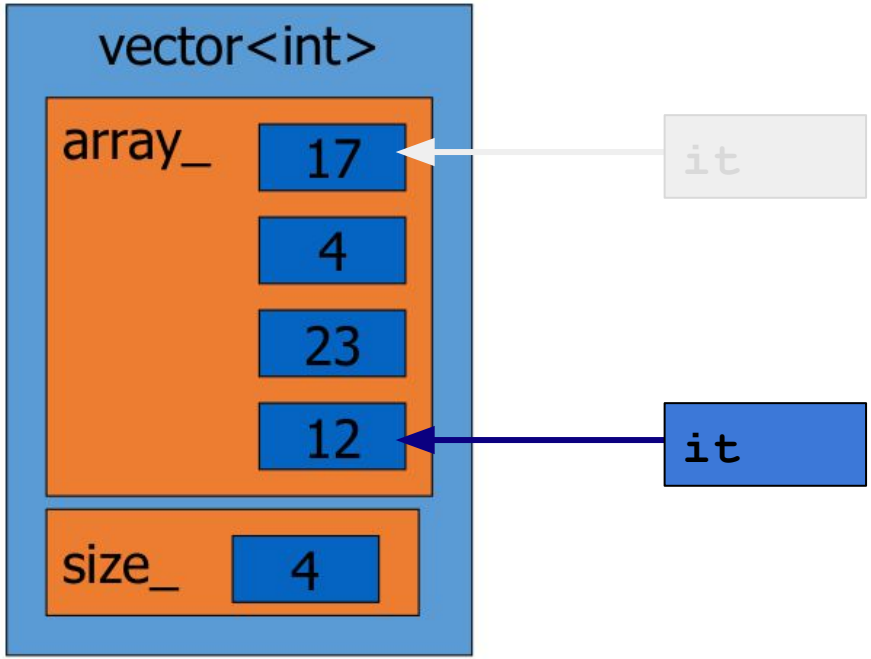
A lato, l'effetto di `it++;`

La differenza è data dal fatto che nel secondo caso viene restituita una copia dell'iteratore nella posizione **prima** dell'incremento.



Scansione

La funzione `advance` permette di eseguire più avanzamenti in una volta. Questo è l'effetto di `advance(it,3)`



Esempio

```
vector<int> v3{ 10, 20, 30, 40};

auto first = v3.begin();
cout << *first << endl;
first++;
cout << *first << endl;
auto last = v3.end()-1;
cout << *last << endl;
```

Esempio, molto utile

```
vector<int> v3{ 10, 20, 30, 40};

for (auto i = v3.begin(), i!=v3.end; ++i)
    cout << *i << endl;
```

Esempio wow!

```
list<int> v3{ 10, 20, 30, 40};

// range-based for loop
for (auto v : v3)
        cout << v << endl;
```

One size fits all.. not!

- Anche se il modello filosofico degli iteratori è attraente, non è totalmente veritiero. Infatti gli iteratori NON SONO TUTTI uguali!
- Alcune strutture supportano vari tipi di operazione. Altre sono più povere, invece, supportano meno operazioni
- Per esempio, le liste non supportano l'accesso diretto. Altre non supportano il "tornare indietro".
- Introduciamo quindi il concetto di "categoria di un iteratore"

Categorie

Tipo di iteratore	Descrizione	Operazioni supportate
Accesso casuale (più ricco)	Salva e recupera valori Muove avanti e indietro Accesso random	* = ++ –> == != –– + – [] < > <= >= += –=
bidirezionale	Salva e recupera valori Muove avanti e indietro	* = ++ –> == != ––
In avanti	Salva e recupera valori Muove solo in avanti	* = ++ –> == !=
input	Recupera valori (ma non li salva) Muove solo in avanti	* = ++ –> == !=
output (più povero)	Salva i valori (ma non li recupera) Move forward only	* = ++

Iteratori & Classi STL

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

Note ulteriori

Finora abbiamo visto gli iteratori applicati ai container STL, quindi come funzioni membro di oggetti.

In realtà si possono applicare anche ad altro, in particolare agli array standard

Esempio

```
#include <iostream>
#include <iterator>
int main() {
 int arr[]={1,2,3,4,5,6,7,8,9,10 };
    for (auto it = std::begin(arr); it != 
std::end(arr); it++)
        std::cout << " " << *it << std::endl;
}
```

Esempio

```
#include <iostream>
#include <iterator>
int main() {
 int arr[]={1,2,3,4,5,6,7,8,9,10 };
    for (auto i : arr)
        std::cout << " " << i << std::endl;
}
```

CHALLENGE TIME!!

E’ tutto per oggi!