

# CoderFarm - Corso base

## Lezione 5

Carlo Collodel, Francesco Cerroni

28 novembre 2022

# Ordinamento efficiente

`std::sort`

Come ordinare un contenitore in ordine crescente?

Uso la funzione `std::sort(it_inizio, it_fine);`

Come ordinare un contenitore in ordine decrescente?

Uso la funzione `std::sort(rit_inizio, rit_fine);`

Come ordinare un contenitore con un altro ordinamento?

Uso la funzione `std::sort(rit_inizio, rit_fine, fun_cmp);`

Si può definire un nuovo modo di ordinare degli elementi riscrivendo l'operatore `<`, questo può servire in caso si voglia definire un ordinamento particolare oppure se il contenitore usa degli elementi per cui non è definito un ordinamento di default.

# Esempi

std::sort standard



```
#include <algorithm>
...

vector<int> vec = {115, 1, 44, 33, 7};
/*
 * ordino il vettore in modo crescente
 * 1 7 33 44 115
 */
sort(vec.begin(), vec.end());

/* ordine decrescente: 115 44 33 7 1 */
sort(vec.rbegin(), vec.rend());
```

# Esempi

std::sort con comparatore custom

```
#include <algorithm>
...
/* decido che gli elementi < 100 sono sempre piu` grandi di quelli >= 100 */
bool comparatore(int left, int right) {
    if (left < 100) {
        return (right < 100 ? left < right : false);
    } else {
        return (right < 100 ? true : left < right);
    }
}

vector<int> vec = {115, 1, 404, 33, 7, 911};

/*
 * ordino con il mio comparatore custom
 * 115 404 911 1 7 33
 */
sort(vec.begin(), vec.end(), comparatore);
```

# Esempi

std::sort con comparatore lambda



```
#include <algorithm>
...

vector<int> vec = {115, 1, 404, 33, 7, 911};

/*
 * ordino decrescente usando una funzione lambda
 * & "cattura" le variabili locali (in questo caso non e` necessario)
 * 115 404 911 1 7 33
 */
sort(vec.begin(), vec.end(), [&](int left, int right) {
    return left > right;
});
```

# Stringhe

Array di caratteri

Come può un computer lavorare su frasi e testi?

Una stringa di testo (un "blocco" di testo) in fin dei conti non è altro che un array di caratteri.


In C++ possiamo dichiarare array di caratteri tramite la sintassi:

```
char frase[] = "testo qui...";
```

Inoltre `frase[i]` è un carattere!

# Stringhe

## Esempi di Array di caratteri



```
/*
 * ogni array di caratteri dovrebbe avere alla fine un carattere
 * terminatore (con valore 0 nella tabella ascii,
 * viene anche detto "NULL character/byte").
 */
char frase[7] = {'a', 'm', 'o', 'g', 'u', 's', '\0'};

/* notare l'assenza del carattere terminatore */
char stessa_frase[7] = "amogus";

/* notare la lunghezza di 11 caratteri, per comprendere il NULL byte */
char altra_frase[11] = "ciao mondo";
```

# Stringhe

Cosa non fare con Array di caratteri!

## La trappola

Gli array di caratteri sono *pur sempre array!*, non si possono concatenare, non posso usare l'operatore di assegnazione (=) per copiare array, **non si possono confrontare con ==**, ...



# Stringhe

Cosa (non) fare con Array di caratteri!



```
/* dichiaro un array di caratteri con 10 elementi */
char frase[10];
frase = "ciao"; /* NON LO POSSO FARE */

char inizio[] = "para";
char fine[] = "cadute";

/* FUNZIONA anche se sono array, l'operatore << e' definito */
cout << inizio << fine << endl;

/* NON FUNZIONA la concatenazione */
char intero[] = inizio + fine;

char inizio_uguale[] = "para";
/* STAMPA 0 */
cout << (inizio == inizio_uguale) << endl;
```

# Stringhe

`std::string`

Esiste qualcosa di più versatile?

Possiamo usare il tipo `std::string`, che supporta tutte le operazioni mostrate prima (e molte altre!).

# Stringhe

## Esempi std::string

```
string frase = "ciao";

frase += " mondo"; /* piu` efficiente di frase = frase + " mondo" */
cout << frase << endl;

frase = "altra frase";
cout << (frase == "altra frase") << endl;

/* stampa 5 caratteri dalla posizione 6 */
cout << frase.substr(6, 5) << endl;

/* converto da stringa a intero */
cout << stoi(string("100")) << endl;

/* converto da double a stringa */
cout << to_string(100.100) << endl;
```

# Stringhe

Alcuni metodi utili

- ▶ `string::operator[]`
- ▶ `string::find(stringa_cercata, pos_partenza)`
- ▶ `string::substr(pos_partenza, lunghezza)`
- ▶ `string::push_back(carattere)`
- ▶ `string::pop_back()`
- ▶ `string::clear()`
- ▶ ...<sup>1</sup>

---

<sup>1</sup>[https://en.cppreference.com/w/cpp/string/basic\\_string](https://en.cppreference.com/w/cpp/string/basic_string)


## std::set

I set permettono di memorizzare un insieme di valori in base al loro ordinamento, ogni elemento viene memorizzato una volta sola (inserire un elemento già inserito non causa nessun cambiamento). Le più importanti operazioni supportate sono:

- ▶ inserire un elemento -  $O(\log N)$
- ▶ eliminare un elemento -  $O(\log N)$
- ▶ cercare se un elemento è presente -  $O(\log N)$
- ▶ accedere a tutti gli elementi **secondo l'ordinamento** -  $O(N)$

# std::set

## Esempi std::set



```
#include <iostream>
#include <set>
using namespace std;
int main() {
    set<int> S; // insieme vuoto

    S.insert(5); // inserisci il valore 5
    S.erase(5); // elimina il valore 5
    if (S.count(3)) { // restituisce 1 se 3 è presente nel set
        cout << "S contiene il valore 3\n";
    }

    cout << "S contiene " << S.size() << " elementi";

    for (int &i: S) { // itera su tutti gli elementi in ordine
        cout << i << " ";
    }
}
```

## std::multiset

Funzionano come i set ma permettono di memorizzare più volte lo stesso valore. La funzione *count* in questo caso restituisce il numero di volte che un valore è presente. La libreria da includere è `<set>`.

## `std::unordered_set` e `std::unordered_multiset`

A differenza dei `set` i valori non vengono salvati in ordine ma le operazioni sono supportate in  $O(1)$  ammortizzato. Esiste anche la versione che può contenere più volte lo stesso valore: *`unordered_multiset`*. Sono contenuti nella libreria `<unordered_set>`.




## std::map

Le mappe permettono di creare delle strutture dati simili agli array, ma come indici possono avere qualsiasi tipo di dato. Ne esistono di 2 tipi:

- ▶ *map*: supportano le operazioni in  $O(\log N)$  ma memorizzano gli indici in maniera ordinata
- ▶ *unordered\_map*: supportano le operazioni in  $O(1)$  ammortizzato ma non conservano gli indici in ordine

# std::map



```
#include <map>
using namespace std;
int main() {
    // mappa che ha stringhe come key e conserva interi
    map<string, int> mappa;

    mappa["ciao"] = 4; // associa a "ciao" il valore 4

    cout << mappa.size() << "\n"; // stampa il numero di key

    // controlla se esiste la key "among us"
    if (mappa.count("among us")) {
        // ...
    }
}
```

## std::stack

Lo stack è una struttura dati rappresentabile come una "pila" di dati che supporta 3 operazioni:

- ▶ aggiungere un elemento in cima
- ▶ rimuovere l'elemento in cima
- ▶ accedere all'elemento in cima

std::stack supporta tutte queste operazioni in  $O(1)$  ammortizzato.

# std::stack



```
#include <stack>
using namespace std;
int main() {
    stack<int> st;

    st.push(5); // aggiungi 5 in cima
    st.push(10);

    cout << "L'elemento in cima è " << st.top() << "\n";

    st.pop(); // rimuovi l'elemento in cima
}
```

# Nearest smaller element

Il problema del nearest smaller element è il seguente:

## Problema

Dato un array di interi determinare per ogni elemento la posizione del primo elemento minore alla sua sinistra. Formalmente sia  $a_0, a_1, \dots, a_n$  la sequenza. Per ogni indice  $i$  bisogna trovare il massimo valore dell'insieme  $X = \{j \in \{0 \dots i-1\} \mid a_j < a_i\}$ .

# Nearest smaller element

Il problema si può risolvere con uno stack:

- ▶ Ad ogni iterazione (da inizio array a fine array) rimuovo dallo stack gli elementi finchè quello in cima è maggiore di quello corrente (e la stack non si svuota)
- ▶ L'elemento ora in cima è il nearest smaller element dell'elemento corrente (se lo stack è vuoto vuol dire che non esiste)
- ▶ Aggiungo allo stack l'elemento corrente e ripeto

# Nearest smaller element

## Esempio

Scegliamo come sequenza  $[1, 3, 4, 2, 5]$ . Alla fine di ogni iterazione lo stack appare così:

0 [1]

1 [1, 3]

2 [1, 3, 4]

3 [1, 2]

4 [1, 2, 5]

# Nearest smaller element

## Implementazione

```

#include <stack>
#include <vector>
using namespace std;
int main() {
    vector<int> v = {1, 3, 4, 2, 5};

    stack<int> st;
    vector<int> nse; // nearest smaller element
    for (int i = 0; i < v.size(); i++) {
        while (st.size() > 0 && st.top() >= v[i]) {
            st.pop();
        }

        if (st.size() == 0) {
            nse[i] = -1; // non esiste, lo segno uguale a -1
        } else {
            nse[i] = st.top();
        }

        st.push(v[i]);
    }
}
```



# Analisi ammortizzata

A volte negli algoritmi non possiamo determinare il numero di passaggi ad ogni iterazione, ma solo quello totale. Infatti nel caso del nearest smaller element sappiamo solo che ogni elemento viene aggiunto e rimosso dallo stack al massimo 1 volta, perciò la complessità della soluzione è  $O(N)$ .

# Esercizi!

- ▶ Tornello olimpico  
<https://territoriali.olinfo.it/#/task/tornello>
- ▶ Antivirus  
<https://territoriali.olinfo.it/#/task/antivirus>
- ▶ Ruota della fortuna  
<https://territoriali.olinfo.it/#/task/fortuna>
- ▶ Tieni aggiornato il catalogo  
<https://training.olinfo.it/#/task/catalogo>
- ▶ La camera dei cestini  
<https://training.olinfo.it/#/task/cestini>

# Fine

Ci vediamo alla prossima lezione!

- ▶ **E-Mail:** `base@coderfarm.it`
- ▶ **Telegram:** T.B.D.