

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Ordinamento e copia

Stefano Ghidoni



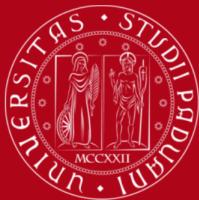
Agenda

- Ordinamento
- Criteri di ordinamento
 - Gestiti mediante lambda expression
- Copia
- Copia con predicato



Function object: criticità

- I function object sono una buona tecnica, ma hanno una debolezza:
 - Definiti in una regione di codice
 - Usati altrove
- È possibile rimuovere questo fenomeno usando un altro strumento:
 - Lambda expression



Lambda expression

- Una lambda expression (o lambda function) è un modo per definire una funzione localmente
- Equivalente a:
 - Definire un function object
 - Crearne uno immediatamente e usarlo come argomento di una funzione
- Tale function object è anonimo e non utilizzabile altrove



La funzione non ha un nome, ma ha un corpo, questo vuol dire che non può essere chiamata fuori dal contesto in cui è definita

sort() e lambda expression



sort()

- La funzione sort() effettua un ordinamento
- Due versioni:
 - Ordinamento con operator<

```
template<typename Ran>
    // Ran deve possedere un random_access_iterator
void sort(Ran first, Ran last);
```

- Ordinamento con funzione dedicata

```
template<typename Ran, typename Cmp>
    // Ran deve possedere un random_access_iterator
    // Cmp deve essere una funzione di ordinamento su Ran
void sort(Ran first, Ran last, Cmp cmp);
```



```
template<typename Ran, typename Cmp>
    // Ran deve possedere un random_access_iterator
    // Cmp deve essere una funzione di ordinamento su Ran
void sort(Ran first, Ran last, Cmp cmp);
```

- Cmp è un comparison function object
 - Deve soddisfare i requisiti di *Compare*
 - Accetta due argomenti a, b
 - Risultato convertibile implicitamente a bool
 - Risultato significa "a compare prima di b" in questo ordinamento
 - *Compare* è un **named requirement**

Cmp deve essere un'entità che esprima il concetto di ordine, altrimenti vi sarà un errore in compilazione



sort()

- sort() ordina tra due iteratori
 - Possiamo quindi ordinare anche solo una parte di un container

```
void test(vector<int>& v)
{
    // ordina la prima metà
    sort(v.begin(), v.begin() + v.size()/2);

    // ordina la seconda metà
    sort(v.begin() + v.size()/2, v.end());
}
```



sort() e container

- Spesso ci interessa ordinare un intero contenitore
- È disponibile la funzione sort() per i contenitori
 - Non è parte della STL
 - Definita in std_lib_facilities.h

```
template<typename C>
    // C è un contenitore
void sort(C& c)
{
    sort(c.begin(), c.end());
}
```



Esempio di ordinamento

- Un esempio con l'algoritmo sort() e una lambda expression
- Cerchiamo di ordinare un vettore di Record:

```
struct Record{  
    string name;  
    char addr[24];  
    // ...  
};
```



Criteri di ordinamento

- Dato un vector di record

```
vector<Record> vr;
```

- Abbiamo due modi di ordinarlo – in base a due diversi predicati:

```
struct Cmp_by_name{
    bool operator()(const Record& a, const Record&b) const
        { return a.name < b.name; }
};

struct Cmp_by_addr{
    bool operator()(const Record& a, const Record&b) const
        { return strcmp(a.addr, b.addr, 24) < 0; }
};
```

Confronto lessicografico



Criteri di ordinamento

- Esistono due criteri di ordinamento perché sono presenti due dati membro
 - Situazione tipica quando si ordina in base ai dati membro

```
//...  
  
sort(vr.begin(), vr.end(), Cmp_by_name());  
  
// ...  
  
sort(vr.begin(), vr.end(), Cmp_by_addr());  
  
// ...
```



Criteri di ordinamento

- Esistono due criteri di ordinamento perché sono presenti due dati membro
 - Situazione tipica quando si ordina in base ai dati membro
- Questa situazione è spesso gestita con le lambda expression
 - Vediamo come



Criteri con lambda expression

```
// ...  
  
sort(vr.begin(), vr.end(),  
    [] (const Record& a, const Record& b)  
        { return a.name < b.name; }  
);  
  
// ...  
  
sort(vr.begin(), vr.end(),  
    [] (const Record& a, const Record& b)  
        { return strncmp(a.addr, b.addr, 24) < 0; }  
);  
  
// ...
```



Lambda expression

- Le lambda expression accettano argomenti come le funzioni
- Possono anche *catturare* le variabili locali

```
void func5(std::vector<double>& v, const double& epsilon) {  
    std::transform(v.begin(), v.end(), v.begin(),  
        [epsilon](double d) -> double {  
            if (d < epsilon) {  
                return 0;  
            } else {  
                return d;  
            }  
        });  
}
```

Cattura di variabili locali

Specifica il tipo ritornato
(opzionale, a meno di
ambiguità)



Lambda expression

- La cattura delle variabili locali rende le lambda expression molto comode

Capture clause	Effetto
[&epsilon, zeta]	Cattura epsilon per riferimento e zeta per valore (copia)
[&] Tutte quelle in scope	Cattura tutte le variabili usate nella lambda expression per riferimento
[=]	Cattura tutte le variabili usate nella lambda expression per valore
[&, epsilon]	Cattura tutte le variabili usate nella lambda expression per riferimento, ma epsilon è per valore
[=, &epsilon]	Cattura tutte le variabili usate nella lambda expression per valore, ma epsilon è per riferimento

copy()



- Effettua una copia 😊

```
template<typename In, typename Out>
    // In: iteratore di input
    // Out: iteratore di output
Out copy(In first, In last, Out res)
{
    while(first != last) {
        *res = *first;
        ++res;
        ++first;
    }
    return res;
}
```

Così si possono copiare elementi da un contenitore di un tipo a un contenitore di un altro



- Copia di una sequenza in un'altra sequenza
 - Possono essere container diversi
- Spetta all'utente verificare che sia disponibile sufficiente spazio a destinazione
 - Politica simile al range check per i vector: non sono effettuate operazioni potenzialmente costose e non sempre necessarie



copy() – utilizzo

- Sorgente: segnalati inizio e fine
- Destinazione: segnalato solo l'inizio

```
void f(vector<double>& vd, list<int>& li)
{
    if(vd.size() < li.size())
        error("Target container too small\n");
    copy(li.begin(), li.end(), vd.begin());
}
```



copy_if()

- Esiste anche la copia con verifica di un predicato
 - Stessa sintassi, con un quarto argomento che specifica il predicato

```
void f(const vector<int>& v)
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(), Larger_than(6));
}
```

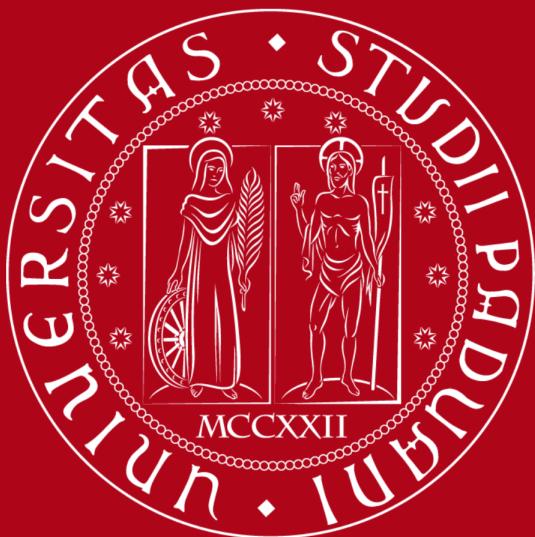


Spunti di approfondimento

- Algoritmi numerici
 - accumulate
 - inner_product
 - partial_sum
 - adjacent_difference



- Ordinamento
- Esprimere un criterio di ordinamento con una lambda expression
- Copia
- Gestione dei contenitori sorgente e destinazione per la copia
- Copia subordinata a un predicato



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Ordinamento e copia

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE