23 - Iteratori

Codice 22-11-2024

```
#include <algorithm>
#include <iostream>

#include <iterator>
#include <string>

////output stream iterator definiti neell'header file iterator, suddiviso in porzioni separate incluse a loro volta

bool lunga (const std::string& s){
    return s.size() >= 10;
}

template< typename InputIt, typename OutputIt, typename UnaryOp>
OutputIt trasform (InputIt first1, InputIt last1,
                   OutputIt d_first, UnaryOp unary_op);


template< class InputIt1, class InputIt2, class OutputIt, class BinaryOp>
OutputIt trasform (InputIt first1, InputIt last1, InputIt first2,
                   OutputIt d_first, BinaryOp binary_op) {//due seq tale che la prima non può essere piu corta della seconda, in modo da lavorare solo sulla sequenza in comune ed avere piu efficienza
    for ( ; first1 != last1; first1++) { //la seconda seq è lunga almeno quanto la prima
        *d_first = binary_op(*first1, *first2);
        ++first2; //x passare al successivo
    }
    return d_first; //ritorno l'operatore di output (quasi tutti gli algo che usano l'operatore di output lo ritornano -> per poter concatenare le operazioni)
}

int main() {
    std::istream_iterator<std::string> first(std::cin);
    std::istream_iterator<std::string> last;

    std::ostream_iterator<std::string> out(std::cout, "\n");

    std::copy_if(first, last, out, lunga);
}
```

Iteratori

Il concetto di iteratore fornisce modi per rappresentare le sequenze, indipendentemente dal tipo utilizzato per l'implementazione.
Esistono 5 categorie che si differenziano per le operazioni supportate:

- **iteratori di input**:
  operazioni consentite

  - `++iter`, avanza di una posizione
  - `iter++`, preferire il prefisso
  - `*iter`, accesso (in read) all'elemento corrente
  - `iter - > m`, equivalente a `(*iter).m` dove si assume che l'elemento abbia tipo classe e che `m` sia un membro della classe
  - `iter1 == iter2`, confronto per uguaglianza tra iteratori (verifica se siamo alla fine della sequenza)
  - `iter1 != iter2` confronto per disuguaglianza
    Esempio: iteratori definiti sugli stream di input `std::istream`, attraverso i quali è possibile leggere i valori sullo stream.

```
#include <iterator>
#include <iostream>

int main() {

  // uso di iteratori per leggere numeri double da std::cin

  std::istream_iterator<double> i(std::cin); // inizio della (pseudo) sequenza
  std::istream_iterator<double> iend;        // fine della (pseudo) sequenza

  // scorro la sequenza, stampando i double letti su std::cout
  for ( ; i != iend; ++i)
    std::cout << *i << std::endl;
}
```

  in questo caso, l'iteratore che indica l'inizio della sequenza si costruisce passando l'input stream (`std::cin`), mentre quello che indica la fine della sequenza si ottiene con costruttore di default.
  Gli iteratori di input non sono riavvolgibili (one shot), ovvero non consentono di scorrere più volte la stessa sequenza, incontriamo un comportamento indefinito. L'operazione di avanzamento consuma l'input letto e per poterlo rileggere occorre salvarlo da qualche altra parte.
- **iteratori forward**:
  effettuano tutte le operazioni degli input iterator, cambia però la semantica. L'incremento non invalida altri iteratori, sono riavvolgibili e consentono di scorrere più volte la stessa sequenza. Se il tipo dell'elemento indirizzato è modificabile, è possibile utilizzarli anche per scrivere sulla sequenza.
  Esempio: iteratori del `std::forward_list`

```
int main() {
  std::forward_list<int> lista = { 1, 2, 3, 4, 5 };

  // Modifica gli elementi della lista
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
  // dell'iteratore usato, che sarebbe std::forward_list<int>::iterator
  for (auto i = lista.begin(); i != lista.end(); ++i)
    *i += 10;

  // Stampa i valori 11, 12, 13, 14, 15
```

```
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
  // dell'iteratore usato, che sarebbe std::forward_list<int>::const_iterator
  for (auto i = lista.cbegin(); i != lista.cend(); ++i)
    std::cout << *i << std::endl;
}
```

//Possiamo usare la copy sugli op del forward list? Si, perche sanno fare tutto ciò che sanno fare gli input iterator (duck type system)

- **iteratori bidirezionali**:
  effettuano tutte le operazioni supportate dagli iteratori forward e consentono di spostarsi all'indietro sulla sequenza con gli operatori di decremento:
  - `- -iter`, si sposta alla posizione precedente
  - `iter - -` (preferire il prefisso)
    Resi disponibili per esempio nel `std::list`, `std::set`, `std::map`...

```
#include <list>
#include <iostream>

int main() {
  std::list<int> lista = { 1, 2, 3, 4, 5 };

  // Modifica gli elementi della lista
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
  // dell'iteratore usato, che sarebbe std::list<int>::iterator
  for (auto i = lista.begin(); i != lista.end(); ++i)
    *i += 10;

  // Stampa i valori all'indietro
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
  // dell'iteratore usato, che sarebbe std::list<int>::const_iterator
  for (auto i = lista.cend(); i != lista.cbegin(); ) {
    --i; // Nota: è necessario decrementare prima di leggere
    std::cout << *i << std::endl;
  }

  // Potevo ottenere (più facilmente) lo stesso effetto usando
  // gli iteratori all'indietro
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
  // dell'iteratore usato, che sarebbe std::list<int>::const_reverse_iterator
  for (auto i = lista.crbegin(); i != lista.crend(); ++i)
    std::cout << *i << std::endl;
}
```

- **iteratori random access**:
  effettuano tutte le operazioni supportate dagli iteratori bidirezionali e consentono di spostarsi liberamente nella sequenza
  operazioni:
  - `iter += n`, sposta iter di n posizioni (avanti: n positivo, indietro: n negativo)
  - `iter -= n`, analogo, ma direzione opposta
  - `iter + n`, calcola un iteratore spostato di n posizioni senza modificare iter
  - `n + iter`, equivalente

- `iter - n`, analogo, direzione opposta
- `iter[n]`, equivalente a `*(iter + n)`
- `iter1 - iter2`, calcola la distanza tra due iteratori, ovvero quanti elementi dividono le due posizioni (definiti sulla stessa sequenza)
- `iter1 < iter2`, true: se iter1 occorre prima di iter2 (sulla stessa sequenza)

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

int main() {
  std::vector<int> vect = { 1, 2, 3, 4, 5, 6 };

  // Modifica solo gli elementi di indice pari
  for (auto i = vect.begin(); i != vect.end(); i += 2)
    *i += 10;

  // Stampa i valori 11, 2, 13, 4, 15, 6
  for (auto i = vect.cbegin(); i != vect.cend(); ++i)
    std::cout << *i << std::endl;
}
```

- **iteratori di output**:
  permettono solamente di scrivere gli elementi di una sequenza, l'operazione di scrittura deve essere fatta una volta sola, successivamente bisogna incrementare per riposizionare l'iteratore e prepararlo per la scrittura successiva.
  operazioni:
  - `++iter`, avanza di una posizione la sequenza
  - `*iter`, accesso in sola scrittura all'elemento corrente
    Non è possibile confrontare tra loro iteratori di output (non necessario).
    Assumono di default che ci sia spazio nella sequenza per scrivere, compito di chi usa l'iteratore garantirlo (! → undefined behavior).

```cpp
#include <iterator>
#include <iostream>

int main() {

  std::ostream_iterator<double> out(std::cout, "\n"); // posizione iniziale
  // Nota: non esiste una "posizione finale"
  // Nota: il secondo argomento del costruttore serve da separatore;
  // se non viene fornito si assume la stringa vuota ""

  double pi = 3.1415;
  for (int i = 0; i != 10; ++i) {
    *out = (pi * i);  // scrittura di un double usando out
    ++out;            // NB: spostarsi in avanti dopo *ogni* scrittura
  }

}
```

Gli iteratori forward, bidirezionali e random access soddisfano tutti i requisiti se gli oggetti "puntati" sono accessibili anche in scrittura.

```
template <typename Iter, typename Iter2>
Iter find_first_of(Iter first, Iter last,
                   Iter2 first2, Iter2 last2) {
    for ( ; first != last; first++) {
        for (Iter2 it2 = first2 ; it2 != last2; it2++){
        if (*first == *it2)
            return first;
        }
    }
    return last;
}
```

Utilizzano gli iteratori forward, non possono utilizzare input iterator in quanto ha bisogno di riavvolgere la sequenza.

```
template <typename ForwardIt, typename T>
void replace(ForwardIt first, ForwardIt last, )
```

replace lavora in lettura e scrittura su una sequenza

```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first);
```

`std::merge` richiede input ordinati per garantire che l'output sia ordinato.

Template di classe `std::iterator_traits`

Alias di tipo per gli iteratori
Quando scriviamo algoritmi generici che sfruttano il concetto di iteratore, abbiamo bisogno di usare nomi canonici per accedere ai tipi associati agli iteratori. Tuttavia, non tutti gli iteratori sono implementati come classi e, quindi, non sempre è possibile accedere direttamente ai loro tipi membri. Per questo motivo, la libreria standard fornisce il template di classe `std::iterator_traits`, che consente di interrogare gli iteratori in modo uniforme, indipendentemente dal fatto che siano classi o tipi primitivi come puntatori.

Un iteratore implementato come classe dovrebbe fornire i seguenti alias di tipo:

- `value_type`: il tipo di dati ottenuto dereferenziando l'iteratore.
- `reference`: il tipo riferimento associato a `value_type`.
- `pointer`: il tipo puntatore associato a `value_type`.
- `difference_type`: un tipo intero con segno che rappresenta la distanza tra due iteratori.
- `iterator_category`: una categoria che indica le operazioni supportate dall'iteratore (input, forward, bidirectional, random access, ecc.).
  Oss.

1. `const_reference` e `const_pointer`, perché è l'iteratore che decide se il `value_type` è o meno in sola lettura;
2. la `iterator_category` è un "tag_type", ovvero un tipo che può assumere un solo valore, il cui unico significato è dato dall'identità del tipo stesso; sono definiti nella libreria standard come:
   - `struct output_iterator_tag { };`
   - `struct input_iterator_tag { };`
   - `struct forward_iterator_tag : public input_iterator_tag { };`
   - `struct bidirectional_iterator_tag : public forward_iterator_tag { };`
   - `struct random_access_iterator_tag : public bidirectional_iterator_tag { }; Le relazioni di ereditarietà dicono, per esempio, che un` bidirection_iterator_tag `può essere convertito implicitamente (up-cast) in un` forward_iterator_tag `o ad un` input_iterator_tag `(un bidirectional è accettabile se viene richiesto un forward, ma non il contrario). Per fornire tutti gli alias di tipo non possiamo utilizzare la tecnica dei contenitori standard, perchè tra gli iteratori ci sono anche tipi che non sono classi, come i puntatori; perciò non possiamo utilizzare:` Iter::value_type `. Aggiriamo il problema usando il template di classe` std::iterator_traits `, in questo modo non interroghiamo direttamente il tipo iteratore, ma la classe traits ottenuta con l'istanziazione del template con quel tipo di iteratore (es.` std::iterator_traits::value_type`).
   Esempio traits:

```
tmp = *iter; //value_type dell'iteratore
typename std::iterator_traits<Iter>::value_type tmp = *iter;
//risolvibile come:
auto tmp = *iter;
```

`std::iterator_traits` è uno degli esempi di uso di classi "traits", ovvero tipi di dato che hanno lo scopo di fornire informazioni ("traits", caratteristiche) di altri tipi di dato. In particolare, consento di effettuare analisi di introspezione anche sui tipi built-in.
Altri esempi:

- `std::numeric_limits` per interrogare i tipi numerici per ottenere informazioni come i valori minimi e massimi rappresentabili, la signedness...
- `std::char_traits` per interrogare i tipi carattere.

Definizione generale

```
template <typename Iter>
struct iterator_traits {
  typedef typename Iter::iterator_category iterator_category;
  typedef typename Iter::value_type        value_type;
  typedef typename Iter::difference_type   difference_type;
  typedef typename Iter::pointer           pointer;
  typedef typename Iter::reference         reference;
};
```

Questa definizione è utilizzata per gli iteratori personalizzati dall'utente:

- il template `iterator_traits` accede ai tipi membri definiti dall'iteratore stesso, come `iterator_category`, `value_type`...

- si assume che il tipo `Iter` (definito dall'utente) abbia già dichiarato questi tipi come membri. In pratica, questo permette a `iterator_traits` di delegare il lavoro di introspezione (cioè determinare i tipi necessari) direttamente all'implementazione dell'iteratore personalizzato.

Quando `Iter` è un puntatore (ad esempio `int*`), la definizione generica non può funzionare, perchè i puntatori non hanno membri come `iterator_category`, `value_type`, ecc. Per la gestione di questi casi si utilizzano le specializzazioni parziali:

Caso 1: Puntatore normale

```
template <typename T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                         pointer;
  typedef T&                         reference;
};
```

- `iterator_category` viene impostata a `random_access_iterator_tag`, perchè i puntatori sono considerati iteratori di tipo "random access" (accesso diretto a qualsiasi elemento con `[]`, `+`, `-`,...);
- `value_type` tipo di dati a cui il puntatore punta (`T`);
- `difference_type` è `ptrfidd_t`, ovvero la differenze tra due puntatori;
- `pointer` è il puntatore stesso (`T*`)
- `reference` è il riferimento al tipo puntato (`T&`).

Caso 2: Puntatore const

```
template <typename T>
struct iterator_traits<const T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef const T*                   pointer;
  typedef const T&                   reference;
};
```