Iteratori

Il concetto di iteratore

È un concetto astratto. Molti algoritmi generici della libreria `standard` lavorano sul concetto di sequenza. Il concetto di iteratore, che prende spunto dal puntatore, fornisce un modo efficace per rappresentare varie tipologie di sequenze, indipendentemente dal tipo concreto usato per la loro implementazione.

Gli iteratori si possono classificare in 5 categorie distinte (che corrispondono, tecnicamente, a 5 concetti correlati ma distinti), che si differenziano per le operazioni supportate e per le corrispondenti garanzie fornite all'utente. Le categorie sono:

- iteratori di input
- iteratori forward
- iteratori bidirezionali
- iteratori random access
- iteratori di output

Torna all'indice

Iteratori di input

Consentono di effettuare le seguenti operazioni:

- `++iter`: avanzamento di una posizione nella sequenza.
- `iter++`: avanzamento postfisso (NON usarlo: preferire la forma prefissa).
- `*iter`: accesso (in sola lettura) 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: tipicamente usato per verificare se siamo giunti al termine di una sequenza.
- `iter1 != iter2`: confronto per disuguaglianza.

Un esempio di iteratore di input è dato dagli iteratori definiti sugli stream di input `std::istream`, attraverso i quali è possibile leggere i valori presenti sullo stream:

```
#include
#include

int main() {

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

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

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

> Nel caso degli istream, 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 col costruttore di default.

Quando si opera con un iteratore di input occorre tenere presente che l'operazione di incremento potrebbe invalidare eventuali altri iteratori definiti sulla sequenza. Per esempio:

```
std::istream_iterator i(std::cin); // inizio della (pseudo) sequenza
auto j = i;
```

```
// ora j e i puntano entrambi all'elemento corrente
std::cout << *i; // stampo l'elemento corrente
std::cout << *j; // stampo ancora l'elemento corrente

++i;             // avanzo con i: questa operazione rende j *invalido*
std::cout << *j; // errore: comportamento NON definito
```

In linguaggio informale, si dice che gli iteratori di input potrebbero essere *"one shot"*; analogamente si dice che potrebbero *"non essere riavvolgibili"* (cioè non consentono di scorrere più volte la stessa sequenza). Intuitivamente, l'operazione di avanzamento *consuma* l'input letto precedentemente (quindi, se lo si volesse rileggere, occorre averlo adeguatamente salvato da qualche altra parte).

Torna all'indice

Iteratori forward

Consentono di effettuare tutte le operazioni supportate dagli iteratori di input. Inoltre, l'operazione di avanzamento effettuata su un iteratore forward *NON* invalida eventuali altri iteratori che puntano ad elementi precedenti nella sequenza (cioè, gli iteratori forward sono riavvolgibili e consentono di scorrere più volte la stessa sequenza).

Infine, se il tipo dell'elemento indirizzato è modificabile, un iteratore forward può essere usato anche per scrivere (non solo per leggere).

Esempi di iteratori forward sono quelli resi disponibili dal contenitore `std::forward_list`:

```
#include
#include

int main() {
 std::forward_list 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::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::const_iterator
 for (auto i = lista.cbegin(); i != lista.cend(); ++i)
     std::cout << *i << std::endl;
}
```

Torna all'indice

Iteratori bidirezionali

Consentono di effettuare tutte le operazioni supportate dagli iteratori forward (e quindi anche tutte quelle degli iteratori di input). Inoltre, consentono di spostarsi all'indietro sulla sequenza, usando gli operatori di decremento:

```
--iter
iter--
```

Esempi di iteratori bidirezionali sono quelli resi disponibili dal contenitore `std::list`. Altri esempi sono gli iteratori resi disponibili dai contenitori associativi (`std::set`, `std::map`, ecc..).

```
#include
#include

int main() {
 std::list 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::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::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::const_reverse_iterator
 for (auto i = lista.crbegin(); i != lista.crend(); ++i)
     std::cout << *i << std::endl;
}
```

Torna all'indice

Iteratori random access

Consentono di effettuare tutte le operazioni supportate dagli iteratori bidirezionali (e quindi anche quelle dei forward e degli input iterator).

Sono inoltre supportate le seguenti operazioni (`n` è un valore intero):

- `iter += n`: sposta iter di n posizioni (in avanti se n è positivo, all'indietro se n è negativo).
- `iter -= n`: analogo, ma sposta nella direzione opposta.
- `iter + n`: calcola un iteratore spostato di n posizioni (senza modificare iter).
- `n + iter`: equivalente a `iter + n`.
- `iter - n`: analogo, ma nella direzione opposta.
- `iter[n]`: equivalente a `*(iter + n)`.
- `iter1 - iter2`: calcola la "distanza" tra i due iteratori, ovvero il numero di elementi che dividono le due posizioni (i due iteratori devono essere definiti sulla stessa sequenza).
- `iter1 < iter2`: restituisce `true` se `iter1` occorre prima di `iter2` nella sequenza (che deve essere la stessa).
- `iter1 > iter2`: analoghi
- `iter1 <= iter2`
- `iter1 >= iter2`

Esempi di iteratori random access sono i puntatori (per esempio sugli *array built-in*) e gli iteratori forniti da `std::vector`, `std::deque`, `std::array`, `std::string`, `std::bitset`, ...

```
#include
#include

int main() {
 std::vector 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;
}
```

Torna all'indice

Iteratori di output

Gli iteratori di output sono iteratori che permettono solamente di scrivere gli elementi di una sequenza: l'operazione di scrittura deve essere fatta una volta sola, dopodiché è necessario incrementare l'iteratore (intuitivamente, per riposizionare correttamente l'iteratore, preparandosi per la scrittura successiva).

Le uniche operazioni consentite sono quindi le seguenti:

- `++iter`: avanzamento di una posizione nella sequenza
- `iter++`: avanzamento postfisso (NON usarlo: preferire la forma prefissa)
- `*iter`: accesso (in sola scrittura) all'elemento corrente

  Si noti che NON viene data la possibilità di confrontare iteratori di output tra di loro, in quanto NON è necessario farlo: un iteratore di output assume che vi sia sempre spazio nella sequenza per potere fare le sue scritture; è compito di chi lo usa fornire questa garanzia e, se la proprietà è violata, si otterrà un undefined behavior.

Un esempio di iteratore di output è dato dagli iteratori definiti sugli stream di output `std::ostream`, attraverso i quali è possibile scrivere valori di un determinato tipo sullo stream.

```
#include
#include

int main() {
 std::ostream_iterator 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
 }
}
```

Si noti che, quando il tipo degli oggetti "puntati" è accessibile in scrittura, gli iteratori forward, bidirezionali e random access soddisfano i requisiti degli iteratori di output e quindi possono essere usati ovunque sia necessario fornire un iteratore di output.

Torna all'indice

Il template di classe iterator_traits

`std::iterator_traits`

Come si è sottolineato, alcune categorie di iteratori implementano un sovrainsieme delle operazioni e garanzie fornite da altre categorie: ciò significa che ogni volta che, nella documentazione di una funzione generica, si afferma che il parametro di tipo `Iter` è richiesto essere (per esempio) un *iteratore forward*, l'utente può istanziare correttamente quel parametro di template usando un qualunque iteratore concreto delle categorie forward, bidirezionale e random access.

L'utente commetterebbe però un errore se istanziasse il parametro `Iter` con uno `std::istream_iterator`, perché questi sono (solo) iteratori di input.

Quando abbiamo visto le interfacce dei contenitori standard, abbiamo notato come essi forniscano un certo numero di alias di tipo che consentono di dare nomi *"canonici"* ad alcuni tipi utili nella definizione e uso dell'interfaccia stessa (`size_type`, `value_type`, `iterator`, ecc.).

La necessità di usare nomi canonici è avvertita anche quando si scrivono algoritmi generici che sfruttano il concetto di iteratore, ragione per cui un iteratore implementato come classe dovrebbe fornire i seguenti type alias:

- `value_type`: tipo ottenuto dereferenziando l'iteratore.
- `reference`: tipo riferimento (al `value_type`).
- `pointer`: tipo puntatore (al `value_type`).
- `difference_type`: tipo intero con segno (per le "distanze" tra iteratori).
- `iterator_category`: un tipo "tag" (marcatore), che indica la categoria dell'iteratore.

Torna all'indice

Osservazioni

Non sono forniti i `const_reference` e `const_pointer`, perché è l'iteratore che decide se il `value_type` è o meno in sola lettura; per esempio, se da un vettore vi di tipo `const std::vector<int>&` estraggo un iteratore usando il metodo `begin()`, otterrò un `std::vector<int>::const_iterator` il cui alias reference è `const int&` e il cui alias pointer è `const int*`.

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). I tipi tag per le categorie di iteratori sono definiti nella libreria standard in questo modo:

```
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 `bidirectional_iterator_tag` può essere convertito implicitamente (tramite up-cast) ad un `forward_iterator_tag` o ad un `input_iterator_tag`, ma *NON* può essere convertito ad un `random_access_iterator_tag`.

Queste conversioni codificano le relazioni esistenti tra le categorie di iteratori, dicendo per esempio che un bidirectional è accettabile quando viene richiesto un forward, ma non vale il viceversa. Questi tag types possono quindi essere usati per codificare versioni alternative di un algoritmo generico scelte in base alla categoria dell'iteratore (come esempi concreti, vedere le funzioni generiche `std::advance` e `std::distance`).

Abbiamo detto che in linea di principio ogni iteratore concreto dovrebbe fornire gli alias di tipo descritti sopra. Ma come farlo? Non possiamo adottare banalmente la tecnica usata per i contenitori standard, perché tra i nostri iteratori ci sono anche tipi che NON sono classi (i puntatori) e che quindi NON consentono di essere interrogati mediante la sintassi che usa l'operatore di scope:

```
Iter::value_type  // con un typename prefisso, se necessario
```

Il problema si risolve usando il template di classe `std::iterator_traits`: invece di interrogare direttamente il tipo iteratore, si interroga la classe traits ottenuta istanziando il template con quel tipo iteratore. Per esempio, se vogliamo conoscere il value type di `Iter`, scriviamo

```
std::iterator_traits::value_type
```

L'uso di `iterator_traits` è solo uno degli esempi di uso di classi *"traits"*, ovvero tipi di dato che hanno lo scopo di fornire qualche informazione (traits, ovvero le caratteristiche) di altri tipi di dato.

In particolare, le classi *traits* consentono di effettuare queste analisi di "introspezione" anche sui tipi built-in (che non forniscono direttamente meccanismi per l'introspezione).

Altri esempi, già intravisti, di classi traits sono:

- Template di classe `std::numeric_limits<T>`: consente di interrogare tipi numerici per ottenere informazioni quali i valori minimi e massimi rappresentabili, la signedness, il fatto di supportare o meno calcoli esatti, ecc.
- Template di classe `std::char_traits<T>`: consente di interrogare i tipi carattere per ottenere accesso, per esempio, alle funzioni di confronto da usare per l'ordinamento lessicografico.

Dal punto di vista dell'utilizzo, il template `std::iterator_traits` è banale. E' però interessante vederne l'implementazione, perché fornisce un esempio semplice di specializzazione *parziale* di template di classe.

Intuitivamente, gli `iterator_traits` devono distinguere i tipi puntatore dagli altri iteratori (di tipo definito dall'utente). Nel secondo caso, che corrisponde al template non specializzato, si *"delega"* al tipo Iter definito dall'utente il compito di

fare il lavoro di introspezione:

```
template
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;
};
```

Quando invece il tipo `Iter` è un puntatore, il template di classe si attiva per restituire all'utente le informazioni che il tipo built-in non sarebbe in grado di fornire. Vengono quindi fornite due specializzazioni parziali (*non-const* e *const*) del template di classe:

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

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

Torna all'indice