Programmazione generica in C++

I template vengono usati in C++ per realizzare il polimorfismo statico:

- Si parla di "*polimorfismo*" in quanto si scrive una sola versione del codice (template) che però viene utilizzata per generare tante varianti (istanze) e quindi può assumere tante forme concrete.
- Il polimorfismo è "*statico*" in quanto la scelta delle istanze da generare avviene staticamente, a tempo di compilazione; cioè non avviene a run-time, come nel caso del polimorfismo "dinamico", che affronteremo in un'altra parte del corso.

La programmazione generica (in C++) è una metodologia di programmazione fortemente basata sul polimorfismo statico (ovvero sui template). Sarebbe però sbagliato pensare che la programmazione generica sia semplicemente basata su definizione e uso di template di classe e funzione: i maggiori benefici della programmazione generica si ottengono solo quando un certo numero di template sono progettati in maniera coordinata, allo scopo di fornire interfacce comuni e facilmente estendibili.

Per comprendere meglio questo punto è utile studiare quella parte della libreria standard del C++ che fornisce i contenitori e gli algoritmi, cercando di capire come questi riescano ad inteargire tra di loro.

Torna all'indice

Contenitori

Un contenitore è una classe che ha lo scopo di contenere una collezione di oggetti (spesso chiamati elementi del contenitore). Essendo spesso richiesto che il tipo degli elementi contenuti sia arbitrario, i contenitori sono tipicamente realizzati mediante template di classe, che si differenziano a seconda dell'organizzazione della collezione di oggetti e delle operazioni fondamentali che si intendono supportare (in maniera efficiente) su tali collezioni.

Torna all'indice

Contenitori sequenziali

I contenitori sequenziali forniscono accesso ad una sequenza di elementi, organizzati in base alla loro posizione (il primo elemento, il secondo, il terzo, ecc.). L'ordinamento degli oggetti nella sequenza non è stabilito in base ad un criterio di ordinamento a priori, ma viene dato dalle specifiche operazioni di inserimento e rimozione degli elementi (effettuati a partire da posizioni determinate della sequenza).

I contenitori sequenziali standard sono:

- Vector
- Deque
- List
- Forward list

Torna all'indice

Vector

```
std::vector < T >
```

Sequenza di `T` di dimensione variabile (a tempo di esecuzione), memorizzati in modo contiguo. Fornisce accesso ad un qualunque elemento in tempo costante. Inserimenti e rimozioni di elementi sono (ragionevolmente) efficienti se fatti in fondo alla sequenza; altrimenti è necessario effettuare un numero lineare di spostamenti di elementi per creare (eliminare) lo spazio per effettuare l'inserimento (rimozione).

È presente un metodo per ottenere un puntatore al primo elemento della sequenza così da permettere l'integrazione con funzioni che lavorano con un puntatore ad un array.

Deque

`std::deque < T >`

Una "double-ended queue" è una coda a doppia entrata, nella quale inserimenti e rimozioni efficienti possono essere effettuati sia in fondo alla sequenza (come nel caso dei vector) che all'inizio della sequenza. Per poterlo fare, si rinuncia alla garanzia di memorizzazione contigua degli elementi (intutivamente, gli elementi vengono memorizzati in "blocchi"). Fornisce accesso ad un qualunque elemento in tempo costante.

List

`std::list < T >`

Sequenza di `T` di dimensione variabile (a tempo di esecuzione), memorizzati (in modo non contiguo) in una struttura a lista doppiamente concatenata. La doppia concatenazione consente lo scorrimento della lista sia in avanti che all'indietro (bidirezionale). Per accedere ad un elemento occorre "raggiungerlo" seguendo i link della lista. Inserimenti e rimozioni possono essere effettuati in tempo costante (nella posizione corrente), perché non occorre spostare elementi.

Forward list

`std::forward_list < T >`

Come una list, ma la concatenazione tra nodi è singola e quindi è consentito lo scorrimento solo in avanti (forward).

Torna all'indice

Pseudo-contenitori

Oltre ai veri contenitori sequenziali, ve ne sono alcuni che sono detto "*pseudo-contenitori*":

- Array
- String
- Bitset

Torna all'indice

Array

`std::array < T,  N >`

Sequenza di `T` di dimensione `N`, fissata a tempo di compilazione.

> Nota: `N` è un parametro valore, non è un typename.

Intuitivamente corrisponde ad un array del linguaggio, ma è immune dalle problematiche relative al type decay e consente di conoscere facilmente il numero di elementi.

String

`std::string`

Può essere visto come una sequenza di caratteri (char).

Nota: `std::string` è un alias per l'istanza `std::basic_string<char>` del template `std::basic_string`; il template si può istanziare con altri tipi carattere, per cui abbiamo gli alias `std::wstring`, `std::u16string` e `std::u32string` per le stringhe di `wchar_t`, `char16` e `char32`.

Bitset

```
std::bitset < N >
```

Una sequenza di esattamente `N` bit.

Nota: `N` è un parametro *valore*, non è un typename.

Torna all'indice

Le operazioni sui contenitori sequenziali

I contenitori sequenziali forniscono:

- costruttori
- operatori per interrogare (gestire) la dimensione
- operatori per consentire l'accesso agli elementi
- operatori per inserire e rimuovere elementi
- operatori di confronto (tra contenitori)
- alcuni altri operatori specifici

È opportuno esaminare in dettaglio le interfacce dei vari contenitori, mettendo in evidenza le somiglianze e le differenze (e magari chiedendosi il motivo di certe differenze). Per farlo, oltre allo studio del libro di testo, è possibile consultare la corrispondente documentazione disponibile online, per esempio ai seguenti indirizzi:

- www.en.cppreference.com
- www.cpplusplus.com

Nota: si rimanda ad un momento successivo l'introduzione ai contenitori associativi della libreria standard.

Torna all'indice

Uno sguardo a std::vector

La dichiarazione nel file header presenta un oggetto di tipo allocator. Questo permette di utilizzare un metodo di allocazione "personalizzata", per gli oggetti rappresentati da `T`. Se non viene specificato verrò utilizzato in automatico quello dello standard.

```
template<
    class T,
    class Allocator = std::allocator
> class vector;
```

Nota: `allocator` è comune a tutti i contenitori. Per sapere di più su std::allocator.

I dati membro:

- `value_type`
- `allocator_type`
- `syze_type`
- `difference_type`
- `reference`
- `const_reference`: reference che permette l'accesso in sola lettura
- `pointer`

- `const_pointer`
- 4 tipi di iteratori (importanti):
    - `iterator`
        - `const_iterator`: permette di iterare sul vector in sola lettura;
        - `reverse_iterator`: fa il contrario di quello che gli viene detto di fare, ad esempio se gli si chiede l'inizio fornisce la fine;
        - `const_revere_iterator`: è un reverse_iterator che permette la sola lettura;

All'interno dei contenitori è presente un costruttore che è considerabile un coltellino svizzero:

```
template
vector(InputIt first, InputIt last,
        const Allocator& alloc = Allocator());
```

Questo permette di inizializzare un contenitore iterando gli elementi compresi tra gli iteratori `first` e `last`.

Esempio:

```
int main(){
 // creo una coda
 std::queue dd;
 /*
 ... popolo dd ...
 */

 // creo un vettore sfruttando il costruttore sopra descritto, iterando tutti gli elementi della coda
 std::vector v(dd.begin(), dd.end());
 return 0;
}
```

Nota: `dd.end()` fa riferimento all'elemento successivo all'ultimo.

Torna all'indice

Algoritmi generici: dai tipi ai concetti

Abbiamo brevemente introdotto quattro contenitori sequenziali standard e (almeno) tre quasi-contenitori: studiandoli in dettaglio, ci siamo probabilmente accorti che sono caratterizzati da interfacce simili, ma non esattamente identiche.

Abbiamo anche notato che le interfacce NON comprendono molti dei servizi che un utente si aspetta di potere utilizzare quando vuole lavorare con collezioni di elementi: per esempio, la classe vector non fornisce un metodo per ordinare gli elementi o per controllare se un elemento con un determinato valore è presente al suo interno.

L'idea di fondo della libreria standard è che questi servizi vengano implementati come "algoritmi generici", all'esterno dei contenitori e in modo il più possibile indipendente rispetto alla specifica implementazione fornita da un determinato tipo contenitore. In altre parole, gli algoritmi generici non sono pensati per lavorare con tipi di dato specifici; piuttosto, sono pensati per lavorare su "concetti" astratti ed essere quindi applicabili a tutti i tipi di dato che forniscono tutte le garanzie che caratterizzano un concetto.

Per rimanere su un esempio concreto, consideriamo un algoritmo che debba cercare un elemento con un certo valore all'interno di un contenitore. A ben pensarci, questo algoritmo non ha una vera necessità di operare su di un tipo contenitore: visto in astratto, l'algoritmo può essere applicato ad una qualunque sequenza i cui elementi possano essere scorsi, dall'inizio alla fine, e confrontati con l'elemento cercato. In altre parole, per questo algoritmo di ricerca, il tipo contenitore può essere sostituito dal "concetto" astratto di sequenza sulla quale si possano fare operazioni di lettura.

Un modo per rappresentare una sequenza dalla quale vogliamo leggere è quello di utilizzare una coppia di iteratori (convenzionalmente chiamati first e last), che servono ad indicare la posizione iniziale della sequenza (first) e la posizione subito dopo l'ultima (last). Si tratta quindi di sequenze "semi-aperte", spesso informalmente indicate con la notazione degli intervalli:

$$[;first,; last;)$$

dove la parentesi quadra iniziale ci informa che l'elemento indicato da first è compreso nella sequenza, mentre la parentesi tonda finale ci informa che l'elemento indicato da last è escluso dalla sequenza.

Torna all'indice

Che cosa è un iteratore?

Un iteratore non è un tipo di dato; è un "concetto" astratto (come il concetto di sequenza).

L'esempio classico di iteratore è dato dal tipo puntatore ad un elemento contenuto in un array: usando una coppia di puntatori, posso identificare una porzione dell'array come la sequenza sulla quale applicare un algoritmo. Il primo puntatore punta al primo elemento della sequenza, il secondo puntatore punta alla posizione immediatamente successiva all'ultimo elemento della sequenza che voglio esaminare.

Esempio che utilizza un iteratore concreto `(int*)`:

```
int* cerca(int* first, int* last, int elem) {
  for ( ; first != last; ++first)
    if (*first == elem)
      return first;
  return last;
}

int main() {
  int ai[200] = { 1, 2, 3, 4, ... };
  int* first = ai;
  int* last = ai + 2; // cerco solo nei primi 3 elementi
  int* ptr = cerca(first, last, 2);
  if (ptr == last)
    std::cerr << "Non trovato";
  else
    std::cerr << "Trovato";
}
```

L'algoritmo di ricerca mostrato sopra funziona solo per i puntatori a interi; per aumentarne l'applicabilità, dovremmo fare la templatizzazione delle funzione "cerca".

Ma in che modo?

Un primo tentativo potrebbe essere quello di effettuare la ricerca usando dei `T*` (puntatori ad un tipo qualunque), nel modo seguente:

```
template
T* cerca(T* first, T* last, T elem) {
  for ( ; first != last; ++first)
    if (*first == elem)
      return first;
  return last;
}
```

ma sarebbe comunque una scelta limitante. Possiamo immaginare che ci possano essere anche altri tipi di dato, oltre ai puntatori, che possano essere usati in modo analogo. Quindi sostituiamo il tipo `T*` con un ulteriore parametro di template che deve fornire il concetto di iteratore (non necessariamente un puntatore):

```
template
Iter cerca(Iter first, Iter last, T elem) {
  for ( ; first != last; ++first)
    if (*first == elem)
      return first;
  return last;
}
```

Quali sono i requisiti per potere istanziare correttamente il tipo `Iter` nell'algoritmo generico qui sopra?

1. `Iter` deve supportare la copia (passato e restituito per valore).
2. `Iter` deve supportare il confronto binario (`first != last`), per capire se la sequenza è terminata o meno.

3. `Iter` deve supportare il preincremento (`++first`), per avanzare di una posizione nella sequenza.
4. `Iter` deve consentire la dereferenziazione (`*first`), per poter leggere il valore "puntato".
5. Il tipo dei valori puntati da `Iter` deve essere confrontabile con il tipo `T` (usando l'operatore ==).

Qualunque tipo di dato concreto, che sia o meno un puntatore, se soddisfa questi requisiti (sintattici e semantici) allora può essere usato per instanziare il mio algoritmo. Si dice che i template applicano delle regole di tipo "strutturali" (in contrapposizione alle regole "nominali"): non importa l'identità del tipo, importa la sua struttura (ovvero le operazioni che rende disponibili e la loro semantica).

Un altro modo di dire le stesse cose (informale e tecnicamente non completamente appropriato) è quello di dire che i template implementano il "duck typing", ovvero un sistema di tipi basato sul "test dell'anatra":

> *"If it walks like a duck and it quacks like a duck, then it must be a duck."*

L'anatra quindi è un concetto astratto: qualunque entità che cammina come un'anatra e fa il verso dell'anatra, è un'anatra.

Essendo specificati usando concetti astratti e non classi concrete, gli algoritmi generici risultano di applicabilità più generale. In particolare, non vi sono algoritmi specifici per un dato contenitore della libreria; piuttosto, ogni contenitore fornisce (attraverso i suoi iteratori) la possibilità di essere visto come una sequenza e gli algoritmi lavorano sulle sequenze.

Vedremo quindi esempi di sequenze (per esempio, in sola lettura, in sola scrittura e in lettura/scrittura), tipicamente rappresentate mediante uno o due iteratori; introdurremo inoltre ulteriori concetti astratti (i callable e alcune varianti più specifiche, come i predicati) che consentirano di astrarre la nozione di "chiamata di funzione", consentendoci di parametrizzare gli algoritmi non solo rispetto alla sequenza, ma anche rispetto alle operazioni da applicare sulla sequenza. L'analisi di questi esempi avrà anche l'utile effetto collaterale di farci prendere confidenza con alcuni degli algoritmi generici della libreria standard.

Torna all'indice

Contenitori associativi

I contenitori associativi sono contenitori che organizzano gli elementi al loro interno in modo da facilitarne la ricerca in base al valore di una "chiave". Abbiamo i seguenti contenitori:

```
std::set
std::multiset
std::map
std::multimap
std::unordered_set
std::unordered_multiset
std::unordered_map
std::unordered_multimap
```

Queste 8 tipologie di contenitori si ottengono combinando (in tutti i modi possibili) tre distinte proprietà:

1. La presenza (o assenza) negli elementi di ulteriori informazioni, oltre alla chiave usata per effettuare le associazioni.

   Se il tipo elemento è formato solo dalla chiave (`Key`), allora abbiamo i contenitori detti "insiemi" (`set`); altrimenti abbiamo i contenitori dette "mappe" (`map`), che associano valori del tipo `Key` a valori del tipo `Mapped`; in particolare, nel caso degli insiemi, il tipo degli elementi contenuti è `const Key`, mentre nel caso delle mappe il tipo degli elementi contenuti è la coppia `std::pair<const Key, Mapped>`.

2. La possibilità o meno di memorizzare nel contenitore più elementi con lo stesso valore per la chiave.

   Nel caso sia possibile memorizzare più elementi con lo stesso valore per la chiave, abbiamo le versioni "multi" dei contenitori (multinsiemi, multimappe, eccetera).

3. Il fatto che l'organizzazione interna del contenitore sia ottenuta mediante un criterio di ordinamento delle chiavi (il tipo `Cmp`) oppure attraverso una opportuna funzione di hasing (il tipo `Hash`).

Nel primo caso, abbiamo la possibilità di scorrere gli elementi nel contenitore in base al criterio di ordinamento; l'implementazione interna deve garantire che la ricerca di un valore con una determinata chiave possa essere effettuata eseguendo un numero di confronti $O(log(n))$, al più logaritmico nel numero n di elementi contenuti (l'implementazione è tipicamente basata su una qualche forma di albero di ricerca bilanciato).

Nel secondo caso (funzione di hashing) si ottengono invece i contenitori "unordered": questi organizzano gli elementi in una tabella hash per cui quando si scorrono non si presentano secondo un criterio di ordinamento "naturale". L'implementazione interna garantisce che la ricerca di un valore con una determinata chiave abbia nel caso medio un costo costante: per fare questo, la funzione di hashing calcola una posizione "presunta" nella tabella hash e poi, usando la funzione di confronto per uguaglianza (il tipo `Equal`) si controlla se vi sono stati clash.

> Eventualmente facendo più confronti fino a raggiungere l'elemento cercato o stabilire che non è presente.

Torna all'indice

Gli adattatori (per contenitori della STL)

Oltre ai contenitori, nella libreria sono forniti gli "adattatori"; questi forniscono ad un contenitore esistente una interfaccia specifica per usarlo "come se" fosse un determinato tipo di dato.

Esempi di adattatori sono `std::stack<T, C>` e `std::queue<T, C>`, che forniscono le classiche strutture dati di pila (LIFO) e coda (FIFO). Al loro interno, usano un altro contenitore standard (il tipo `C`; la scelta di default è `std::deque<T>`, sia per le pile che per le code).

Esiste anche l'adattatore `std::priority_queue<T, C, Cmp>` per le code con priorità (la classica struttura dati heap), nelle quali la priorità tra gli elementi è stabilita dal criterio di confronto `Cmp`. In questo caso, il contenitore `C` usato per default è uno `std::vector<T>`.

> NOTA BENE: gli adattatori **NON** implementano il concetto di sequenza; in particolare, **NON** forniscono i tipi iteratore e i corrispondenti metodi `begin()` e `end()`.

Torna all'indice