5. Template e Libreria Standard

In questo capitolo si vedono i concetti base della programmazione generica e in specifico l'utilizzo di **template**.
Si approfondirá il concetto di template vedendo alcuni strumenti a supporti di essi come i **contenitori** e gli **iteratori**.

Template di Funzione

Costrutto del linguaggio C++ che consente di defenire uno schema (o modello) parametrico per una funzione.
Vediamo un esempio:

```
// Dichiarazione pura di un template di funzione.
template <typename T>
T max(T a, T b);

// Definizione di un template di funzione.
template <typename T>
T max(T a, T b) {
     return (a > b) ?  a :  b;
}
```

- **Parametri**
  Prendendo l'esempio, `T` é un parametro di template: il parametro viene dichiarato essere un `typename` nella lista dei parametri del template. La lista puó contenere un numero arbitrario di parametri separati da virgole. Per convenzione si usano nomi maiuscoli per i parametri di tipo.

- **Istanziazione**
  Dato un template, é possibile generare da esso piú funzioni mediante il meccanismo di istanziazione, che fornisce un **argomento** (della tipologia corretta) ad ogni parametro del template.
  É possibile evitare la sintassi esplicita lasciando al compilatore il compito di dedurre il tipo partendo da quelli passati alla funzione.
  Ad esempio:

  ```
  char c1, char c2;
  int m = max(c1, c2);
  ```

  Si noti che il tipo di `m` (`int`) non influisce sul processo di deduzione.
  Alcune volte il processo di deduzione potrebbe fallire a causa di ambiguitá.

Ad esempio:

```
double d, int i;
int m = max(d, i); // Errore:  é ambiguo.
int m = max<int>(d, i); // Dichiarazione esplicita.
```

É opportuno sottolineare la differenza tra template di funzione e funzione.
In particolare, bisogna capire che un template di funzione NON é una funzione; va visto piú come un generatore di funzioni.
É l'istanza del template, invece, che va a rappresentare la funzione vera e propria.
Vediamo ora un esempio completo di utilizzo di template di funzione in C++.

```
template <typename T>
class GenericCalc {

     public:
          static T sum(T arg1, T arg2) {
               return arg1 + arg2;
          }

          static T sub(T arg1, T arg2) {
               return arg1 - arg2;
          }

          static T div(T arg1, T arg2) {
               return arg1 / arg2;
          }

          static T mul(T arg1, T arg2) {
               return arg1 * arg2;
          };
}


int main() {

     int i1 = 2;
     int i2 = 6;
     int isum = GenericCalc<int>::sum(i1, i2);
     std::cout << i1 << " + " << i2 << " = " << isum;

     float f1 = 2.6f;
     float f2 = 6.8f;
```

```
    float fsum = GenericCalc<float>::sum(f1, f2);
    std::cout << f1 << " + " << f2 << " = " <<fsum;

    return 0;
}
```

- **Specializzazione esplicita**
  Puó capitare che la definizione del template non sia adeguata a tutti i casi di interesse. Il codice potrebbe infatti restituire un risultato ritenuto sbagliato in base ai parametri ricevuti.
  Il template che abbiamo visto `max`, ad esempio, se istanziato con un tipo `const char*` restituirebbe come risultato il massimo dei due puntatori passati, quando invece molto probabilmente l'utente voleva compiere un confronto lessicografico tra due stringhe tipo C.
  Per questo é possibile fornire una definizione **specializzata** per gli argomenti sensibili.

  ```
  // Specializzazione esplicita per T = const char* del template.
  template<>
  const char* max<const char*>(const char* a, const char* b) {
       return strcmp(a, b) > 0;
  }
  ```

  La lista vuota dei parametri (`<>`) indica che si tratta di una specializzazione totale.

- **Istanziazioni Esplicite**
  Una definizione di istanziazione esplicita blucrea e dichiara un elemento (che puó essere una classe, una funzione, ecc...) partendo da un modello (ovvero il template) senza peró utilizzarlo ancora.
  In C++ la definizione di istanziazione esplicita é preceduta dalla parola chiave **extern**.

  ```
  extern template float max(float a, float b);
  ```

  In questo modo si informa il compilatore che, quando verrá usata quell'istanza del template, NON dovrá produrre la corrispondente definizione dell'istanza (evitando cosí la generazione del codice).
  Intuitivamente, `extern` indica che il codice corrispondente dovrá essere trovato dal linker esternamente a questa unitá di traduzione, cioé in un object file generato dalla compilazione di un'altra unitá di traduzione.

  In pratica, la dichiarazione di istanziazione esplicita impedisce che vengano effettuate le istanziazioni implicite cosí da diminuire i tempi di compi-

lazione e/o generare object file più piccoli.

Ordinamento Parziale

Se denotiamo con `istanze(x)` l'insieme di tutte le possibili istanze del template `X`, allora si dice che il template di funzione `X` é piú specifico del template di funzione `Y` se `istanze(x)` é sottoinsieme di `istanze(y)`.

I template dotati dello stesso nome e visibili nello stesso scope sono ordinati principalmente rispetto a questa relazione di specificitá.
Nella risoluzione dell'overloading, le istanze dei template piú specifici sono preferite a quelle dei template meno specifici.

Template di Classe

Un template di classe é un costrutto del C++ che permette di scrivere un modello (schema) parametrico per una classe.
Sono applicabili quasi tutti i concetti visti per i template di funzione, ma con alcune importanti differenze.
L'esempio di implementazione é il seguente.

```
// Dichiarazione pura di un template di classe.
template <typename T>
class Stack;

// Definizione di un template di classe.
template <typename T>
class Stack {
     public:
          /* ...  */
          void push(const T& t);
          void pop();
          /* ...  */
}
```

Per questo tipo di template é fondamentwale distinguere tra il nome del template (`Stack` in questo esempio) e il nome della specifica istanza (`Stack<std::string>`).
Infatti per i template di classe NON si applica la deduzione dei parametri: la lista di argomenti va dichiarata obbligatoriamente.

```
Stack <int> s1; // Istanza implicita del tipo Stack<int>.
Stack s2 = s1; // Errore:  non viene dedotto il tipo T = int.
auto s2 = s1; // Usando "auto" viene applicata la deduzione.
```

L'unico caso in cui é lecito usare il nome del template per indicare il nome della classe mediante istanziazione é all'interno dello scope del template di classe stesso.

```
template <typename>
class Stack {
     /* ...  */
     // Qui gli usi di Stack sono abbreviazioni (lecite) di Stack<T>.
     Stack& operator = (const Stack&);
     /* ...  */
};
```

- **Istanziazione on demand**
  Quando si istanzia implicitamente una classe templatica, vengono generate solo le funzionalitá necessarie per il funzionamento del codice che causa l'istanziazione.
  Questo vuol dire che la dichiarazione di una classe `Stack<int>` non comporta immediatamente l'istanziazione dei metodi `Stack<int>::push(const int&)` e `Stack<int>::pop()`, ad esempio.
  Verrano invece istanziati nel momento in cui andranno a servire e verrano quindi utilizzati.

  In questo modo posso usare un sottoinsieme delle funzionalitá della classe istanziandola con argomenti corretti anche se questi risulteranno non corretti per altre funzionalitá.

  Tuttavia questo tipo di istanziazione ci obbliga, quando vengono scritti i test per la classe templatica, a fornire un insieme di test che copra tutte le funzionalitá di interesse.

- **Specializzazioni parziali**
  Si tratta di specializzazioni di template che sono applicabili non per una scelta specifica degli argomenti (come nel caso delle specializzazioni totali), ma per sottoinsiemi di tipi.
  Una specializzazione parziale di un template (di classe), quindi, è ancora un template di classe, ma di applicabilità meno generale.
  Questo tipo di specializzazioni NON sono supportate dai template di funzioni.

Compilazione dei Template

Il processo di compilazione dei template richiede che lo stesso codice sia analizzato dal compilatore in (almeno) due contesti distinti:

1. Al momento della definizione del template.
2. Al momento dell'istanziazione del template.

Questo significa che, quando si scrive un programma, la definizione di un template deve esser disponibile in tutti i punti del programma dove se ne richiede l'istanziazione.
Generalmente, questo viene effettuato includendo le definizioni (con le eventuali dichiarazioni delle funzioni membro annesse) prima di ogni loro uso nell'unitá di traduzione.

Nell'esercitazione sulla templatizzazione della classe `Stack`, ad esempio, era stato effettuato questo passaggio spostando le definizioni dei metodi della classe all'interno dell'header file `Stack.hh`.

Inoltre questo processo di compilazione puó richiedere, in alcune occasioni, di dover modificare il codice di implementazione dei template per fornire al compilatore le informazioni tali da evitare errori di compilazione.

Polimorfismo Statico

I template in C++ vengono utilizzati per realizzare il cosiddetto polimorfismo statico.
Si parla di **polimorfismo** in quanto si scrive una sola versione del codice (template) che viene utilizzata per generare tante varianti a seconda dei casi (istanze).
Diciamo **statico**, invece, poiché la scelta delle istanze da generare avviene a tempo di compilazione (e non a run-time).

La **programmazione generica** é una metodologia di programmazione basata sul polimorfismo statico (ovvero sui template).
Essa é ottimizzata quando un certo numero di template sono progettati in maniera coordinata, offrendo delle interfacce comuni e facilmente estendibili.

Per questo torna utile vedere la parte della libreria standard del C++ che fornisce gli strumenti di supporto ai template come contenitori e algoritmi generici per capire come questi interagiscono tra di loro.

Contenitori

In generale, un contenitore é una classe che ha lo scopo di contenere una collezione di oggetti (detti elementi del contenitore).
Siccome viene richiesto che il tipo degli elementi sia arbitrario, i contenitori sono tipicamente realizzati mediante template di classe, che si differenziano in base all'organizzazione della collezione di oggetti e in base alle operazioni che si intendono supportare su tali collezioni.

Contenitori Sequenziali

Questo tipo di contenitori fornisce l'accesso ad una sequenza di elementi organizzati in base alla loro posizione.
L'ordine di questi elementi NON é stabilito a priori, bensí viene dato dalle operazioni specifiche di inserimento e rimozione.
I contenitori sequenziali standard sono:

- `std::vector<T>`
  Sequenza di dimensione variabile (run-time) di elementi `T` memorizzati in modo contiguo.
  Fornisce l'accesso a qualunque elemento in tempo costante.
  Inserimenti e rimozioni sono efficienti se fatti in fondo alla sequenza, altrimenti é necessario effettuare un numero lineare di spostamenti per creare (o eliminare) lo spazio necessario per l'inserimento (o la rimozione).

- `std::deque<T>`
  Coda a doppia entrata (double-ended queue) nella quale inserimenti e rimozioni vengono effettuati in maniera efficiente sia in testa che in coda alla sequenza, sacrificando la memorizzazione contigua (memorizzazione a blocchi).
  Fornisce accesso a qualunque elemento in tempo costante.

- `std::list<T>`
  Sequenza di dimensione variabile (run-time) di `T` elementi memorizzati (in maniera non contigua) in una struttura a lista doppiamente concatenata.
  In questo modo si puó consentire lo scorrimento della lista sia in avanti che all'indietro (**bidirezionale**).

  Per accedere a un elemento occorre raggiungerlo seguendo i link della lista.

  Inserimento e rimozione sono effettuati sempre in tempo costante poiché

non occorre spostare elementi.

- `std::forward_list<T>`
  Analoga alla `std::list<T>` ma la concatenazione tra nodi é singola e viene di conseguenza consentito solo lo scorrimento in avanti (**forward**).

Ne esistono anche di altre tipologie che peró non rappresentano dei veri e propri contenitori, ma possiamo chiamarli **pseudo-contenitori**.

- `std::array<T, N>`
  Sequenza di `N` elementi di tipo `T` (dove `N` é un parametro valore, non un typename).

  Corrisponde ad un array del linguaggio ma senza problematiche relative al type decay e che consente di sapere facilmente il numero di elementi.

- `std::string`
  Sequenza di caratteri di tipo char.
  Si puó istanziare il template anche con altri tipi di carattere utilizzando un alias diverso (ad esempio `std::u32string` per le stringhe di tipo char32).

- `std::bitset<N>`
  Sequenza di `N` bit, anche in questo caso `N` é un parametro valore e non un typename.

Ovviamente i contenitori sequenziali forniscono delle **operazioni** base come i costruttori, operatori per gestire la dimensione, operatori di inserimento e di rimozione, oltre ovviamente ad alcuni operatori specifici per il contenitore.

Iteratori a Supporto degli Algoritmi Generici

Un iteratore NON é un tipo di dato, é un concetto astratto rappresentato da un'entitá preposta alla scansione di un contenitore secondo le sue caratteristiche e le sue modalitá di accesso agli elementi, rispettandone i vincoli come la dimensione.

Un esempio tipico é dato dal puntatore ad un elemento contenuto in un array: usando una coppia di puntatori posso identificare una porzione dell'array specifica su cui voglio applicare l'algoritmo.

Il primo puntatore punta al primo elemento della sequenza, il secondo alla posizione immediatamente successiva all'ultimo elemento della sequenza che voglio esaminare.

```
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 colo nei primi 3 elementi.
      int* ptr = cerca(first, last, 2);

      if(ptr == last) {std::cerr << "Non trovato";};
      else {std::cerr << "Trovato";};
}
```

Algoritmi Generici

Abbiamo visto che le interfacce fornite dai contenitori sono molto simili, ma non identiche.
Queste interfacce, peró, NON comprendono molti servizi che l'utente si aspetta di poter utilizzare per una determinata collezione di elementi.

La libreria standard riesce a implementare questi servizi tramite gli **algoritmi generici**, che quindi non sono pensati per lavorare su un tipo di dato specifico, ma sono pensati per lavorare su concetti astratti applicabili a tutti i tipi di dato che ne soddisfano i requisiti.

Ad esempio, l'algoritmo di ricerca visto in precedenza funziona solo con puntatori a interi.
Per renderlo piú versatile occorre effettuare una templatizzazione della funzione `cerca` nel seguente modo:

```
template <typename Iter, typename T>
```

```
Iter cerca(Iter first, Iter last, T elem) {

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

Bisogna quindi rispettare i vari requisiti per poter istanziare correttamente il tipo `Iter` nell'algoritmo generico.

Il tipo `Iter` deve:

1. Supportare la copia.
2. Supportare il confronto binario per vedere se la sequenza é terminata o meno.
3. Supportare il preincremento per avanzare di posizione nella sequenza.
4. Consentire la dereferenziazione per poter leggere il valore "puntato".
5. Il tipo dei valori puntati da `Iter` deve essere confrontabile con il tipo T usando l'operatore ==.

Qualsiasi tipo di dato concreto se soddisfa questi requisiti puó essere usato per istanziare l'algoritmo generico.

Questi algoritmi, quindi, non offrono un'applicabilitá specifica per un determinato contenitore, offrono un'applicabilitá generale: ogni contenitore viene quindi visto come una sequenza su cui gli algoritmi possono lavorare.

Contenitori Associativi

Contenitori che organizzano gli elementi al loro interno in modo da facilitarne la ricerca in base al valore di una **chiave**.
Ci sono 8 tipi di contenitori associativi:

1. `std::set<Key, Cmp>`
2. `std::multiset<Key, Cmp>`
3. `std::map<Key, Mapped, Cmp>`

4. `std::multimap<Key, Mapped, Cmp>`
5. `std::unordered_set<Key, Hash, Equal>`
6. `std::unordered_multiset<Key, Hash, Equal>`
7. `std::unordered_map<Key, Mapped, Hash, Equal>`
8. `std::unordered_multimap<Key, Mapped, Hash, Equal>`

Se il tipo di elemento é formato solo dalla chiave (Key), allora abbiamo un contenitore di tipo `set` (insiemi).
Altrimenti abbiamo i contenitori di tipo `map` (mappe), che associano valori di tipo Key a valori di tipo Mapped.
Nel caso invece in cui sia possibile memorizzare piú elementi con lo stesso valore per la chiave, abbiamo i contenitori `multi` (multi-insiemi).
E infine, se l'organizzazione del contenitore é ottenuta mediante un criterio di ordinamento dell chiavi dato da una opportuna funzione di hash, abbiamo i contenitori `unordered`.

Adattatori

Nella libreria standard sono forniti anche degli **adattatori**, oltre ai contenitori.
Questi forniscono ad un contenitore giá esistente un'interfaccia specifica per usarlo come fosse un determinato tipo di dato.

Esempi di adattatori sono `std::stack<T, C>` e `std::queue<T, C>` che forniscono le strutture dati classiche pila (LIFO) e coda (FIFO).

Template Type Deduction

La **template type deuction** (ovvero la deduzione dei tipi per i parametri di template) é il processo messo in atto dal compilatore per semplificare l'istanziazione (esplicita o implicita che sia) e la specializzazione esplicita di template di funzione.

Questa forma di deduzione serve al programmatore per evitare di scrivere in modo ripetuto (che puó anche essere soggetto a sviste) la lista degli argomenti da associare ai parametri del template di funzione.

Supponiamo quindi di avere la seguente dichiarazione di funzione templatica:

```
template <typename TT>
void f(PT param);
```

Il compilatore dunque, di fronte alla chiamata di funzione `f(expr)` usa il tipo di `expr` per dedurre:

1. Un tipo specifico `tt` per `TT`.
2. Un tipo specifico `pt` per `PT`.

Questo causa l'istanziazione del template di funzione che porta alla seguente funzione:

```
void f<tt>(pt param);
```

Il processo di deduzione si distingue in tre casi che vediamo di seguito.

- **Universal reference**
  Si ha universal reference quando abbiamo l'applicazione di && al nome di un parametro typename del mio template di funzione.
  Ad esempio se TT é il parametro del typename:

  ```
  TT&& // É un universal reference.
  const TT&& // NON é un universal reference (r-value reference).
  std::vector<TT>&& // NON é un universal reference (r-value reference).
  ```

  Il nome **universal** sta a indicare che, nonostante venga usata la sintassi per i riferimenti a r-value, puó essere dedotto per `PT` un riferimento sia a r-value che a l-value a seconda del tipo `te` dell'espressione.
  Assumiamo:

  ```
  int i = 0;

  // Esempio per rvalue.
  f(5); // te = int, deduco pt = int&&, tt = int.

  // Esempio per lvalue;
  f(i); // te = int&, deduco pt = int&, tt = int&.
  ```

- **PT puntatore o riferimento**
  Si effettua un pattern matching tra il tipo `te` e `PT` ottenendo i tipi `tt` e `pt`.
  Ad esempio:

```
template <typename TT>
void f(TT* param);

f(&i); // te = int*, deduco pt = const int*, tt = int.
```

Per il caso dei riferimenti l'esempio é analogo:

```
template <typename TT>
void f(TT* param);

f(i); // te = int&, deduco pt = int&, tt = int.
```

- **PT né puntatore né riferimento**
  In questo caso, siccome si ha un passagio per valore, argomento e parametro sono due oggetti distinti.
  Di conseguenza, eventuali riferimenti e qualificazioni `const` dell'argomento NON si estendono al parametro.

Auto Type Deduction

Nel linguaggio (C++11) é stata introdotta la possibilitá di definire le variabili utilizzando la parole chiave **auto**, senza dover specificare il tipo e lasciando di conseguenza al compilatore il compito di dedurlo a partire dall'espressione usata per inizializzare la variabile.

Vediamo qualche esempio base.

```
auto i = 5; // i ha tipo int.
const auto d = 5.3; // d ha tipo const double.
auto ii = i * 2.0 // ii ha tipo double.
const auto p = "Hello" // p ha tipo const char* const;
```

La auto type deduction segue (in larga misura) le stesse regole della template type deduction.
In pratica, quando si osserva una definizione del tipo:

```
auto& ri = ci;
```

In questo caso la parola chiave `auto` svolge il compito del parametro template `TT`, la sintassi `auto&` corrisponde a `PT` e l'inizializzazione `ci` corrisponde all'espressione.

Iteratori

Come abbiamo visto, 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 l'implementazione.

Gli iteratori si possono classificare in cinque categorie in base alle operazioni supportate e alle garanzie fornite all'utente.

- **Iteratori di input**
  Consentono di effettuare le seguenti operazioni:

  - `++iter`
    Avanzamento di una posizione nella sequenza.

  - `*iter`
    Accesso in sola lettura all'elemento corrente.

  - `iter->m`
    Accesso in sola lettura quando l'elemento ha tipo classe e `m` é un membro della classe.

  - `iter1 == iter2`
    Confronto per uguaglianza tra iteratori, usato tipicamente per verificare se é terminata la sequenza.

  - `iter1 != iter2`
    Confronto per disuguaglianza.

  Un esempio standard di iteratore di input é dato dagli iteratori definiti sugli stream di input (`std::istream`), attraverso i quali é possibile leggere i valori presenti sullo stream.
  Ad esempio:

```
#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 si ottiene col costruttore di default.

Quando si opera con un iteratore di input bisogna tener conto che l'operazione di incremento potrebbe invalidare altri iteratori definiti sulla sequenza. Infatti, l'operazione di avanzamento **consuma** l'input letto e se lo si volesse rileggere bisognerebbe averlo salvato opportunamente da qualche parte.

- **Iteratori di output**
  Permettono solamente di scrivere gli elementi di una sequenza; dopodiché bisogna incrementare l'iteratore per posizionarlo correttamente per la scrittura successiva.

  Intuitivamente, le uniche operazioni consentite sono:

  - `++iter`
    Avanzamento di una posizione nella sequenza.

  - `*iter`
    Accesso in sola scrittura all'elemento corrente.

  Non vengono forniti operatori di confronto tra iteratori poiché non é necessario farlo: si assume che ci sia sempre spazio nella sequenza per effettuare una scrittura.
  Sará compito di chi lo usa fornire questa garanzia che, se violata, genererá un undefined behavior.

- **Iteratori Forward**
  Consentono di effettuare tutte le operazioni degli iteratori di input ma l'operazione di avanzamento (incremento) NON invalida eventuali altri iteratori che lavorano sulla sequenza.

Quesro vuol dire che sono iteratori riavvolgibili che permettono di scorrere piú volte la stessa sequenza.
Un esempio di iteratore forward é quello fornito dal contenitore `std::forward_list`.

```
#include <forward_list>
#include <iostream>

int main() {

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

      // Modifica degli elementi della lista.
      for(auto i = lista.begin(); i != lista.end(); ++i)
            *i += 10;
      // L'uso di "auto" in questo caso mi ha risparmiato dallo
      // scrivere il tipo dell'iteratore che
      // sarebbe std::forward_list<int>::iterator.

      // Stampa i valori 11, 12, 13, 14, 15.
      for(auto i = lista.cbegin(); i != lista.cend(); ++i)
            std::cout << *i << std::endl;
      // auto = const_iterator.
}
```

- **Iteratori bidirezionali**
  Implementano tutte le operazioni degli iteratori forward e in piú consentono di spostarsi all'indietro sulla sequenza usando gli operatori di decremento `--iter` e `iter--`.
  Esempi di iteratori bidirezionali sono quelli del contenitore `std::list`.

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

int main() {

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

      // Modifica degli elementi della lista.
      for(auto i = lista.begin(); i != lista.end(); ++i)
            *i += 10;

      // Stampa dei valori all'indietro.
      for(auto i = lista.cbegin(); i != lista.cend(); ) {
            --i; // É necessario decrementare prima di
            // effettuare l'operazione.
```

```
            std::cout << *i << std::endl;
        }
        // auto = const_iterator.
}
```

- **Iteratori random access**
  Consentono di effettuare tutte le operazioni degli iteratori bidirezionali aggiungendone anche altre (con `n` valore di tipo intero):

  - `iter += n`
    Sposta iter di n posizioni (in avanti con n positivo, all'indietro con n negativo).

  - `iter -= n`
    Analogo ma sposta nella direzione opposta.

  - `iter + n`
    Calcola un iteratore spostato di n posizioni senza modificare iter.

  - `iter - n`
    Analogo ma nella direzione opposta.

  - `iter[n]`
    Equivale a *(iter + n).

  - `iter1 - iter2`
    Calcola la distanza tra due iteratori della stessa sequenza.

  - `iter1 < iter2`
    Restituisce vero se iter1 occore prima di iter2 nella sequenza. Ci sono analoghi per gli altri operatori di confronto.

Template di Classe Iterator Traits

In linea di principio, ogni iteratore fornisce alcuni **alias** per dare nomi canonici ad alcuni tipi utili (come ad esempio `value_type` ottenuto deferenziando l'iteratore).

Tuttavia non si puó utilizzare la tecnica usata per i contenitori standard poiché tra gli iteratori ci sono anche tipi che non sono classi (come i puntatori) e che quindi non consentono di essere interrogati con la sintassi che usa l'operatore di scope:

```
Iter::value_type
```

Usando il template di classe **iterator traits** (`std::iterator_traits`) possiamo scavalcare questo problema e interrogare direttamente il tipo iteratore: si interroga la classe traits ottenuta istanziando il template con quel tipo iteratore. Ad esempio se vogliamo conoscere il value type di `Iter` scriviamo:

```
std::iterator_traits<Iter>::value_type
```

Questo template di classe é solo uno degli esempi di uso di classi traits che hanno il compito di fornire informazioni (traits, ovvero le caratteristiche) su altri tipi di dato.

Callable

Molti degli algoritmi generici resi diponibili dalla libreria standard si presentano in due diverse versioni, ad esempio:

1. ```
   template <typename FwdIter>
   FdwIter adjacent_find(FwdIter first, FwdIter last);
   ```

   In questa versione il predicato binario per il controllo di equivalenza che viene usato é `operator==`.

   Diverse istanze del template possono usare definizioni diverse (overloading) dell'operatore, peró il nome della funzione di equivalenza rimane fissato.

2. ```
   template <typename FwdIter, typename BinPred>
   FdwIter adjacent_find(FwdIter first, FwdIter last, BinPred pred);
   ```

   Questa versione é parametrizzata rispetto a una determinata **policy**.
   Consente quindi di utilizzare un qualunque tipo di dato fornito dall'utente a condizione che si comporti come predicato binario sugli elementi della sequenza.

Ci chiediamo ora quali sono i tipi di dato concreto che sono ammessi come parametro dalla funzione pred e che ci permettono di compilare correttamente il test.

```
if(pred(*first, *next)) {...}
```

Considerando un caso meno specifico, in queste situazioni ci servirá sapere quali tipi di dato `Test` possono essere utilizzati (ai proprio valori `test`) come nomi di funzione in una chiamata.

```
test(arg1, ..., argN)
```

L'insieme di questi tipi di dato forma il concetto di **callable**:

1. Puntatori a funzione.
2. Oggetti funzione
3. Espressioni lambda (dal C++11).

Puntatori a Funzione

Negli esempi concreti visti finora, i parametri collable sono sempre stati istanziati usando un opportuno puntatore a funzione.
Ad esempio:

```
bool pari(int i);
```

Per instanziare il predicato unario (in questo caso) della `std::find_if`, il parametro typename `UnaryPred` viene legato al tipo concreto `bool (*)(int)` che non é altro che un puntatore a una funzione che prende come argomento un intero per valore e restituisce un booleano.

Oggetti Funzione

Oltre alle funzioni vere e proprie, vi sono altri tipi di dato i cui valori possono essere invocati come le funzioni rispettando cosí i requisiti del concetto collable.
In particolare, una classe che fornisce una definizione del metodo `operator()` consente ai suoi oggetti di essere utilizzati al posto delle vere funzioni nella sintassi della chiamata.

Vediamo un esempio.

```
struct Pari {
     bool opeartor()(int i) const {
          return i % 2 == 0;
     }
};

int foo() {
     Pari p;
     if(p(345)) {...}
     ...
}
```

L'oggetto `p` (di tipo struct `Pari`) in sé NON é una funziona ma, essendo fornito di un metodo `operator` puó essere invocato come se la fosse.

Gli oggetti funzione, da un punto di vista tecnico, vengono utilizzati per ottenere vantaggi in termini di efficienza rispetto alle normali funzioni; in particolare forniscono al compilatore piú opportunitá per ottimizzare il codice.

Vediamo un esempio specifico: si consideri un programma che lavora su vettori di interi e istanzia piú volte la funzione generica `std::find_if` per effettuare ricerche nel vettore usando criteri di ricerca diversi (predicati unari).
Prendiamo ora i seguenti predicati unari espressi mediante funzione:

```
bool pari(int i);
bool positivo(int i);
bool numero_primo(int i);
```

Tutte queste tre funzioni prendono come argomento un intero e restituiscono un booleano, di conseguenza quando si istanzia l'algoritmo `std::find_if` sul vettore si ottiene sempre la stessa istanza del template di funzione:

```
// Per comoditá usiamo i seguenti alias di tipo.
using Iter = std::vector<int>::iterator
using Ptr = bool (*)(int);

// Quindi la specifica istanza ottenuta sará questa.
Iter std::find_if<Iter, Ptr>(Iter first, Iter last, Ptr pred);
```

Viene generato un codice unico che deve riconoscere e gestire opportunamente le tre possibili invocazioni.
La chiamata al predicato é implementata come chiamata di funzione.

Se invece implementassimo i tre predicati mediante oggetti funzione (definendo

quindi tre classi distinte), quando si istanzia l'algoritmo `std::find_if` sul vettore si otterrano tre diverse istanze del template di funzione:

```
Iter std::find_if<Iter, Pari>(Iter first, Iter last, Pari pred);
Iter std::find_if<Iter, Positivo>(Iter first, Iter last, Positivo pred);
Iter std::find_if<Iter, Numero_Primo>(Iter first, Iter last, Numero_Primo
pred);
```

Qui quando viene generato il codice per una delle istanze, il compilatore vede l'invocazione di uno solo dei tre metodi `operator()` e puó quindi ottimizzare il codice per quella specifica invocazione.

Espressioni Lambda

Espressioni utilizzate come comoda sintassi abbreviata che definisce un oggetto funzione **anonimo** (nel senso che viene definito a fronte di un unico punto del codice che lo invoca) e immediatamente utilizzabile.

Vediamo subito un esempio di istanziazione di `std::find_if` con una lambda expression che implementa il predicato `pari` su di un tipo `T`.

```
void foo(const std::vector<long>& v) {

      auto iter = std::find_if(v.begin(), v.end(),
      [](const long& i) { return i%2 == 0; });
      // Utilizzo di Iter ...
}
```

In questo caso l'espressione lambda é data dalla sintassi:

```
[](const long& i) { return i%2 == 0; }
```

Dove abbiamo i seguenti elementi:

1. `[]` - Capture list (lista delle catture).

2. `(const long& i)` - Lista dei parametri (opzionale).

3. `{ return i%2 == 0; }` - Corpo della funzione.

Questo uso della lambda expression corrisponde intuitivamente alle seguenti operazioni:

1. Definizione di una classe anonima per oggetti funzione dotata di un nome univoco scelto dal sistema.
2. Definizione (all'interno della classe) di un metodo `operator()` che ha parametri, corpo e tipo di ritorno specificati (o dedotti) dall'espressione lambda.
3. Creazione di un oggetto funzione "anonimo", avente il tipo della classe anonima, da passare a `std::find_if`.

In pratica é come se avessimo scritto il seguente codice:

```
struct Nome_Univoco {
     bool operator()(const long& i) const { return i%2 == 0; }
};

auto iter = std::find_if(v.begin(), v.end(), Nome_Univoco());
```

La lista delle catture puó essere usata quando l'espressione lambda deve accedere a variabili locali visibili nel punto in cui viene creata.
Ad esempio:

```
void foo(const std::vector<long>& v, lon soglia) {

     auto iter = std::find_if(v.begin(), v.end(),
     [soglia](const long& i) { return i>soglia; });
     // Utilizzo di Iter ...
}
```

Si possono catturare piú variabili utilizzando come separatore la virgola, e si possono catturare per valore (`[=soglia]`, che é quella base sottointesa nell'esempio) oppure per riferimento (`[&soglia]` utile per evitare copie costose).

Il metodo `operator()` é qualificato `const` e di conseguenza le variabili catturate hanno accesso in sola lettura. Se si vuole consentire la modifica, bisogna aggiungere alla lambda il modificatore **mutable**.

```
[soglia](const long& i) mutable {
     soglia = 100;
     return i>soglia;
}
```