C++ Standard Library

- Array
- Vettore
- Liste
- Mappe
- Iteratori e algoritmi
- Analisi costo computazionale
- Codice di esempio
  - https://github.com/egalli64/corso-cpp folder b7

Concetti chiave

- Basata sulla Standard Template Library (STL) di Alexander Stepanov, tre concetti
  - Container (collezione, aggregato, sequenza)
  - Iteratore (astrazione del concettto di puntatore)
  - Algoritmo
- Sviluppati in modo
  - efficiente (seguendo l'impostazione che viene dal C)
  - parametrico – via template, programmazione generica
  - ortogonale – semplice associazione tra i costrutti di base
- Nell'interfaccia di un container ci aspettiamo
  - Costruttori (default, copy, move, …) e distruttore
  - Metodi per operare sugli elementi: accesso, inserimento (*), eliminazione
    - Oltre all’inserimento “classico”, da C++11 anche via *emplacement*, che permette di operare ottimizzazioni
  - Iteratori

Array

- Il concetto di array che arriva dal C è di livello molto basso, in cambio offre una estrema efficienza
  - allocato su stack (dimensione fissata a compile-time) o heap – in C++ allocato/deallocato via new[] / delete []
  - L'indirizzo del primo elemento di un array corrisponde all'indirizzo dell'array
- Nella C++ Standard Library, **std::array** wrappa il concetto di array C sullo stack
  - Richiede come variabile meta del template il tipo e la dimensione
  - Il costruttore di default non inizializza la memoria
  - Il costruttore variadico permette di inizializzare tutti gli elementi, implicitamente a zero
- Dimensione di un array: size_t size()
- Accesso a un elemento: T& operator[](size_t)
- Iteratori al primo e ultimo elemento (mutabile o costante) : begin(), end() – cbegin(), cend()
- Assegnamento di un valore a tutti gli elementi: void fill(const T&)
- Primo e ultimo elemento: T& front(), T& back()

Vettore

- std::vector è tipicamente la prima scelta tra i container, più flessibile di std::array
  - Wrappa un array C su heap, dando l'illusione che la sua dimensione sia dinamica
- Tra i ctor
  - Default, crea un vettore vuoto
  - Un parametro di tipo size_t, alloca per la dimensione indicata
  - Dimensione e un valore, alloca e inizializza
  - Variadico, alloca e inizializza con diversi valori
- Tra i metodi (oltre a quelli visti per std::array, quando disponibili)
  - Cambio della dimensione: void resize(size_t) e void resize(size_t, const T&)
  - Aggiunta di un elemento in ultima posizione: push_back(), emplace_back()
  - Aggiunta di un elemento in una data posizione: insert(), emplace()

Liste

- Lista doppiamente linkata, std::list
- Lista semplicemente linkata, std::forward_list
  - Introdotta in C++11
  - Funzionalità ridotte al minimo essenziale

Mappe

- Note anche come array associativi o dizionari
  - Gestiscono relazioni chiave-valore, il valore è acceduto normalmente via chiave
- std::map = chiavi sono organizzate in un albero binario (BST red-black)
  - Ordinate, accesso in tempo logaritmico
- std::unordered_map = chiavi sono organizzate in una hash table
  - Disordinate, accesso in tempo costante
- Normalmente, accediamo alle coppie chiave-valore
  - Struttura pair, con componenti first (chiave) e second (valore)

Iteratori

- Categorizzati in cinque gruppi, dal più debole al più forte
  - Un algoritmo dovrebbe cercare di usare l'iteratore più debole possibile, in modo da poter essere usato su più container
- I due iteratori più deboli, a singola passata e monodirezionali, sono
  - **input**: sola lettura, richiede gli operatori
    - ++ (passaggio all'elemento successivo)
    - * e -> (dereferenziamento dell'iteratore per accesso al valore)
    - == e != (ad es., poter determinare se si è raggiunta la fine del container)
  - **output**: sola scrittura
- **forward**: lettura e scrittura, anche lui monodirezionale
  - Ma la sequenza su cui opera deve essere accedibile anche dopo il suo uso
- **bidirectional**, simile al forward, ma può moversi in due direzioni
  - Richiede anche l'operatore --
- **random**, può accedere a ogni elementi del container in tempo O(1)
- Il range-for, (parte del linguaggio e non della libreria standard) può operare anche su array raw
  - Sostanzialmente richiede un iteratore input

Algoritmi

- Agli algoritmi di accesso sequenziale e di ricerca basta un input iterator
  - iterator find(iterator, iterator, const T&)
    - Richiede due iteratori che delimitano la ricerca (aperto a destra) e il valore cercato
    - Ritorna l'iteratore all'elemento trovato, o end()
  - int accumulate(iterator, iterator, T)
    - Richiede due iteratori che delimitano l'intervallo d'azione (aperto a destra) e il valore di partenza
    - Ritorna il valore accumulato dei valori trovati
- Un algoritmo di lettura e scrittura ha bisogno di almeno un forward iterator
  - void replace(iterator, iterator, const T&, const T&)
    - Gli iteratori delimitano l'intervallo (aperto a destra) su cui cercare il valore da sostituire con un nuovo valore
    - La modifica è fatta in-place
- Un algoritmo che necessita un bidirectional iterator
  - void reverse(iterator, iterator)
    - Richiede due iteratori che delimitano l'intervallo d'azione (aperto a destra), modifica in place il container, ribaltandolo

Algoritmi

- Gli algoritmi di sorting nella Standard Library richiedono accesso casuale agli elementi
  - Obbligatorio l'uso di un iteratore random, con limitazione del campo d'uso
  - Richiedono due iteratori che delimitano l'intervallo d'azione (aperto a destra)
  - Ritornano void, per efficienza l'ordinamento è fatto in-place
    - void sort(iterator, iterator) // di solito una versione di Quick Sort
    - void stable_sort(iterator, iterator) // di solito basato su Merge Sort
      - Se ci sono più elementi considerati uguali dall'algoritmo, mantengono l'ordine relativo
- Due esempi di Programmazione Funzionale negli algoritmi
  - iterator find_if(iterator, iterator, Predicate)
    - Richiede l'intervallo d'azione (aperto a destra) e un predicato: funzione, functor o lambda che ritorna un booleano
    - Ritorna l'iteratore al primo elemento che soddisfa il predicato, o end()
  - Function for_each(iterator, iterator, Function)
    - Richiede l'intervallo d'azione (aperto a destra) e una funzione, functor o lambda, eseguita per ogni elemento
    - Ritorna la funzione passata come parametro

Complessità degli algoritmi

- Caso migliore, peggiore, medio in tempo e spazio
  - In funzione del numero “n” di elementi su cui l’algoritmo opera
- “O grande”, limite asintotico superiore della funzione
  - Costante O(1)
  - Logaritmica O(log n)
  - Lineare O(n)
  - Linearitmico O(n log n)
  - Quadratica O(n^2) – Polinomiale O(n^c)
  - Esponenziale O(c^n)
  - Fattoriale O(n!)

Complessità degli algoritmi