PROGRAMMAZIONE AVANZATA

4 Contenitori e oggetti composti

Contenitori

- La libreria standard del C++ definisce molte classi **contenitore**
- L'istanza di una classe contenitore serve per alloggiare **collezioni di elementi**, in particolare collezioni di **oggetti**.

I contenitori si differenziano per:

- **cosa** possono contenere (elementi di tipo specifico o di qualunque tipo)
- **come** si accede agli elementi (per **posizione**, per **chiave**)
- efficienza di alcune **operazioni critiche** (lettura, scrittura, inserimento, ...)

Contenitore	**Contenuto**	**Accesso**	**Operazioni efficienti**
`string`	caratteri	posizione	accesso + (inserim./canc. alla fine)
`vector`	qualsiasi	posizione	accesso + (inserim./canc. alla fine)
`list`	qualsiasi	posizione	accesso/inserim./canc. a inizio e fine
`map`	tipo ordinato	chiave	inserimento/cancellazione/ricerca

Vettori e stringhe

Vettori

- `std::vector<T>` è il tipo dei **vettori** di elementi di tipo T
- la **dimensione** del vettore può variare nel tempo (oggetto **mutabile**)
- accesso per **posizione** $i \in [0, n)$ dove n è la dimensione del vettore

```
std::vector<int> x;   // vettore di numeri interi
std::vector<float> y; // vettore di numeri con virgola
std::vector<Jar> z;   // vettore di giare
```

Stringhe

- `std::string` è il tipo delle **stringhe** (= sequenze di caratteri)
- simile a `std::vector<char>`, ma con operazioni specifiche

```
std::string s = "Un saluto";
```

Operazioni su vettori e stringhe

Operazione	Descrizione
`c.size()`	Restituisce il numero di elementi in `c`
`c.resize(n,x)`	Cambia la dimensione di `c` a `n`, usando `x` per gli eventuali elementi aggiunti
`c.push_back(x)`	Aggiunge `x` a `c` dopo l'ultimo elemento
`c.pop_back()`	Rimuove l'ultimo elemento di `c`
`s.erase(i,n)`	Rimuove `n` caratteri a partire dalla posizione `i`
`c[i]`	Riferimento all'`i`-esimo elemento di `c`, senza controllo
`c.at(i)`	Riferimento all'`i`-esimo elemento di `c`, con controllo

- `c` è un contenitore di tipo `std::string` o `std::vector<T>`
- `s` è un contenitore di tipo `std::string`
- `x` è un elemento di tipo `char` o `T`, a seconda del tipo di `c`
- `i` ed `n` sono numeri interi
- I riferimenti possono essere usati per leggere e scrivere elementi

Esempio: eliminazione di spazi

```
void elimina_spazi_inefficiente(std::string& s) {
  int i = 0;
  while (i < s.size())
    if (isspace(s[i])) s.erase(i, 1); else i++;
}
```

- **Vantaggio** (trascurabile): non creo altre stringhe
- **Svantaggi**: codice inefficiente e con effetti globali

```
std::string elimina_spazi(const std::string& s) {
  std::string r; // inizialmente vuota
  for (int i = 0; i < s.size(); i++)
    if (!isspace(s[i])) r.push_back(s[i]);
  return r;
}
```

- **Vantaggi**: codice efficiente e con effetti locali
- **Svantaggio** (trascurabile): creo e restituisco una nuova stringa

Esempio: rimario

Problema

- Dato un **vocabolario** (elenco di parole), trovare tutte quelle che **finiscono** con un suffisso dato

Rappresentazione dei dati

- Parole e suffissi sono rappresentati come **stringhe**
- Un vocabolario è rappresentato come un **vettore di stringhe**

Note

- La rappresentazione scelta non è necessariamente la più efficiente per il problema in questione, ma ignoriamo questi aspetti a favore della semplicità del codice

`#include` e definizione dei tipi

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

typedef std::vector<std::string> vocabolario;
```

Note

- `fstream` definisce classi/metodi per leggere/scrivere file
- Il tipo del vocabolario è `std::vector<std::string>`
- Con `typedef` definiamo un **sinonimo** per questo tipo, dandogli un nome più breve e più significativo (= migliore leggibilità del codice)
- `std::vector<std::string>` e `vocabolario` sono **intercambiabili**

Lettura del vocabolario

```
vocabolario leggi_vocabolario() {
  vocabolario d;
  std::ifstream f("italiano.txt");
  while (f.good()) {
    std::string parola;
    f >> parola;
    d.push_back(parola);
  }
  return d;
}
```

Note

- Usiamo la classe `std::ifstream` per **leggere** un file
- Il nome del file è un argomento del costruttore della classe
- Il metodo `good` restituisce `true` se il file è stato aperto correttamente e ci sono ancora informazioni (= parole) da leggere
- `d.push_back` inserisce ogni parola letta dal file in fondo al vocabolario
- Il file `italiano.txt` è fornito nell'archivio insieme alle slide

Ricerca delle rime

```
bool termina_con(const std::string& p,
                 const std::string& s) {
  int k = p.size(); // lunghezza della parola
  int l = s.size(); // lunghezza del suffisso
  return k < l ? false : p.substr(k - l) == s;
}

void cerca_rime(const vocabolario& d,
                const std::string& s) {
  for (int i = 0; i < d.size(); i++)
    if (termina_con(d[i], s))
      std::cout << "  " << d[i] << std::endl;
}
```

Note

- `s.substr(n)` restituisce il **suffisso** di `s` a partire dal carattere in posizione `n`
- Versioni recenti del C++ forniscono un metodo `ends_with` il cui effetto è analogo a quello della funzione `termina_con`

Il rimario completo

```
int main() {
  vocabolario d = leggi_vocabolario();
  std::cout << "Ho " << d.size()
            << " parole nel vocabolario " << std::endl;
  std::string suffisso;
  do {
    std::cout << "\nFa rima con (^C o Q per uscire): ";
    std::cin >> suffisso;
    if (suffisso ≠ "Q") cerca_rime(d, suffisso);
  } while (suffisso ≠ "Q");
}
```

Nota

- ≠ è l'operatore classico di **disuguaglianza** !=

Esempio: vettori in spazi vettoriali

Vettore come oggetto di tipo `std::vector<T>`

- Ha una dimensione n (variabile) ed n elementi (di tipo T)
- Posso leggere/scrivere i singoli elementi
- Posso **inserire**/**rimuovere** elementi

Vettore come elemento di uno spazio vettoriale

- Ha una dimensione n (costante) ed n elementi
- Posso leggere (ma non scrivere) i singoli elementi
- Posso **sommare** vettori e calcolare il **prodotto scalare** di vettori

In sintesi

- Ci sono analogie tra vettori (C++) e vettori (di uno spazio vettoriale)...
- ...ma le due entità supportano operazioni differenti
- Possiamo **adattare** i vettori C++ per modellare vettori di uno spazio

Adattamento di oggetti

```
class Vector { // vettore in uno spazio vettoriale
private:
  std::vector<int> data; // contenitore degli elementi
public:
  Vector(int);
  int size() const;
  int operator[](int) const;
  int& operator[](int); // serve per l'inizializzazione
  int mul(const Vector&) const;
};
```

- Usiamo `std::vector<int>` come **contenitore** di elementi di un vettore
- Forniamo le **operazioni tipiche** di un vettore di uno spazio vettoriale

Nota

Definiamo come membri gli operatori [ ] per leggere/scrivere elementi di un vettore. La possibilità di scrivere elementi è una **deroga necessaria** per poter inizializzare vettori. Versioni recenti del C++ hanno un meccanismo conveniente per inizializzare contenitori (vedi initializer list).

Costruttore e metodi di accesso

```
Vector::Vector(int n) {
  if (n < 0) throw std::domain_error("invalid size");
  data.resize(n, 0);
}

int Vector::size() const {
  return data.size();
}

int Vector::operator[](int i) const { // per leggere
  return data.at(i);
}

int& Vector::operator[](int i) {      // per scrivere
  return data.at(i);
}
```

- Il controllo della validità di `i` in `get` e `set` è **delegato** a `at`
- Notare l'uso di `&` nel tipo di ritorno e l'assenza di `const` nel secondo metodo `[]` per permettere la modifica degli elementi del vettore

Prodotto scalare di vettori

```
int Vector::mul(const Vector& v) const {
  if (size() ≠ v.size())
    throw std::domain_error("invalid size");
  int res = 0;
  for (int i = 0; i < size(); i++)
    res += data[i] * v.data[i];
  return res;
}
```

Esempio: calcolo del modulo di un vettore

```
  Vector v(2);
  v[0] = 1;
  v[1] = 2;
  double m = std::sqrt(v.mul(v)); // m = ||v||
```

Esercizi

1. Scrivere una funzione `inserisci_spazi(s)` che inserisca in `s` uno spazio dopo ogni virgola che non è già seguita da uno spazio. Fornire due versioni della funzione, una che modifica `s` (analoga a `elimina_spazi_inefficiente`) e l'altra che restituisce una nuova stringa con il risultato (analoga a `elimina_spazi`).
2. Aggiungere alla classe `Vector` metodi corrispondenti alle seguenti operazioni: moltiplicazione di un vettore per uno scalare, somma e sottrazione di vettori.
3. Realizzare una classe `Matrix` per rappresentare una matrice di numeri interi. Dotare la classe di metodi corrispondenti alle seguenti operazioni: lettura del numero di righe/colonne; lettura di una riga; lettura di una colonna; lettura/scrittura di un elemento della matrice; trasposta; moltiplicazione per scalare; somma/sottrazione; moltiplicazione matriciale; potenza. **Suggerimento**: Usare un campo con tipo `std::vector<Vector>` per memorizzare il contenuto della matrice per righe. Per agevolare la realizzazione dei metodi può essere opportuno dotare la classe di altri campi. **Nota**: Questo esercizio è importante ma non banale. La soluzione è fornita, ma verrà comunque ripresa e discussa in seguito.