Strutture Dati e Iteratori

Q: Quando usare `vector` vs `list`?

A:

- `vector`: struttura dati sequenziale con accesso casuale O(1). Preferibile nella maggior parte dei casi per:
  - Migliori performance di accesso
  - Località dei dati in memoria
  - Minor overhead di memoria
- `list`: lista doppiamente collegata. Usare solo se necessario per:
  - Frequenti inserimenti/rimozioni in mezzo alla sequenza
  - Necessità di splicing
  - Iteratori che devono rimanere validi dopo modifiche

NB: tutto parte di `std`! Noi lo richiamiamo ogni volta

Q: Come si implementa correttamente un ciclo con iteratori?

A: Pattern standard:

```
for(std::vector<T>::iterator it = v.begin(); it != v.end(); ++it) {
    // Accesso all'elemento: *it
}

// Con const_iterator per vettori const:
for(std::vector<T>::const_iterator cit = v.cbegin(); cit != v.cend(); ++cit)
{
    // Accesso in sola lettura: *cit
}
```

Si usa end() perché è ad accesso casuale, si ricordi questo; altrimenti, si opta per `auto` (non toglie punti, anzi, basta usarlo bene).

```
for(auto it = v.begin(); it != v.end(); ++it) {
    // Accesso all'elemento: *it
}
```

Q: Come gestire la rimozione di elementi durante l'iterazione?

A:

1. Pattern corretto per rimozione sicura:

```
for(auto it = v.begin(); it != v.end();) {
    if(condizione_rimozione) {
        delete *it;  // Se necessario deallocare
        it = v.erase(it);  // erase() ritorna iteratore al prossimo elemento
    } else {
        ++it;
    }
}
```

2. Errori comuni da evitare:

- Non incrementare l'iteratore dopo erase()
- Non utilizzare l'iteratore dopo erase() senza riassegnazione

Gestione della Memoria e Const-Correctness

Q: Come gestire correttamente il const in parametri e ritorni?

A:

- Parametri:
  - `const vector<T>&`: per passaggio read-only
  - `vector<T>&`: per modifiche alla collezione
  - `const T* const`: puntatore costante a dato costante
  - `const T*`: puntatore a dato costante
- Ritorni:
  - `vector<T>`: per copia
  - `const vector<T>&`: per riferimento read-only
  - `vector<T>&`: per riferimento modificabile (raro)

Q: Come gestire la deallocazione della memoria?

A:

1. Pattern corretto:

```
// Deallocazione + rimozione
if(ptr) {
    delete ptr;  // Prima dealloco
    ptr = nullptr;  // Poi invalido il puntatore
}
v.erase(it);  // Infine rimuovo dalla collezione
```

2. Punti chiave:

- Sempre verificare ptr != nullptr prima di delete
- Sempre nullificare i puntatori dopo delete
- Deallocare prima di erase() per evitare memory leak

Type Checking e Cast

Q: Quando usare typeid vs dynamic_cast?

A:

- `typeid`:
  - Per verificare uguaglianza esatta del tipo dinamico
  - Esempio: `typeid(*ptr) == typeid(DerivedClass)`
- `dynamic_cast`:
  - Per verificare se un oggetto è di un tipo o sottotipo
  - Ritorna nullptr se il cast fallisce (con puntatori)
  - Esempio: `if(dynamic_cast<DerivedClass*>(ptr))`

Q: Come gestire i cast?

A:
Pattern comuni:

```
vector<const Tipo* const> v; // se abbiamo più const PROBABILE che faremo solo degli inserimenti....

// (1) Conversione al tipo non costante
auto* derived = dynamic_cast<DerivedClass*><const_cast<BaseClass>(base))

// (2) Conversione al tipo costante
auto* derived = dynamic_cast<const DerivedClass*>(base);

// Se devo CANCELLARE l'oggetto RIMUOVO il const -> const_cast
// Se devo SOLO INSERIRLO, posso TENERE il const
```

Gestione delle Eccezioni

Q: Quali sono i pattern comuni per le eccezioni?

A:

1. Tipi di eccezioni:

- Classi custom (es: `class CustomException {};`)
- Tipi standard (`std::exception`, `std::string`)
- Tipi della gerarchia (es: lanciare oggetto di tipo `C`)

2. Pattern di lancio:

```
// Verifica condizione e lancio
if(error_condition) {
    throw CustomException();
}

// Controllo nullptr
if(!ptr) {
    throw std::string("nullptr");
}

// Controllo conteggio
if(count > max_allowed) {
    throw LimitExceededException();
}
```

Note sulla Gerarchia IOS

Q: Come gestire oggetti della gerarchia IOS?

A:

- Verifica stato:

```
if(!stream.good()) // Verifica tutti i bit di errore
if(stream.fail()) // Verifica fallimento operazione
```

- Reset stato:

```
stream.clear() // Reset tutti i bit
stream.setstate(ios::goodbit) // Set stato specifico
```

- Posizione:

```
stream.tellg() // Posizione lettura
stream.tellp() // Posizione scrittura
```

Q: Come gestire i file streams?

A:

```
if(auto* fs = dynamic_cast<fstream*>(stream)) {
    if(fs->is_open()) {
        fs->close();
    }
}
```

Si ricordi che negli include:

- sstream è utile per stringstream
- fstream serve per fstream

Note sulla Gerarchia QT

Q: Come gestire widget QT?

A:
Pattern comuni:

```
// Ridimensionamento
widget->resize(size);

// Clonazione
widget->clone();

// Verifica stato
button->isDown();

// Cast sicuro tra widget
if(auto* slider = dynamic_cast<QSlider*>(widget)) {
    slider->setSliderDown(true);
}
```

Normalmente, la classe base della gerarchia è QWidget oppure QAbstractButton e si tratta di convertire tra i singoli tipi

Referenziazione e Dereferenziazione

Q: Come gestire correttamente referenziazione e dereferenziazione di puntatori e iteratori?

A:

1. Dereferenziazione di puntatori:

```
T* ptr = new T();
*ptr;  // Accede al valore puntato
(*ptr).method();  // Chiama un metodo sull'oggetto puntato
ptr->method();  // Sintassi alternativa equivalente a (*ptr).method()
```

2. Dereferenziazione di iteratori:

```
vector<T>::iterator it = v.begin();
*it;  // Accede all'elemento corrente
(*it).method();  // Chiama un metodo sull'elemento
it->method();  // Sintassi alternativa equivalente
```

3. Reference (&):

```
T& ref = obj;  // Reference a obj
ref.method();  // Opera direttamente su obj
```

4. Pattern comuni:

```
// Con puntatori
T* ptr = &obj;  // & prende l'indirizzo di obj
*ptr = value;   // * dereferenzia il puntatore

// Con iteratori
auto it = v.begin();
*it = value;    // Modifica l'elemento corrente
it->field = value;  // Modifica un campo dell'elemento
```

Se il mio container è di tipo `Tipo*` allora:

```
// Con puntatori
vector<Tipo*> v;

Sottotipo* s = dynamic_cast<Sottotipo*>(v[i]);

v.push_back(s);
```

Se il mio container è di tipo `Tipo` allora:

```
// Con puntatori
vector<Tipo> v;

Sottotipo* s = dynamic_cast<Sottotipo*>(*v[i]);
```

```
v.push_back(*s);
```

Q: Quali sono gli errori comuni da evitare?

A:

1. Dereferenziazione di nullptr:

```
T* ptr = nullptr;
*ptr;  // Errore: segmentation fault!

// Pattern corretto:
if(ptr) {
    *ptr = value;
}
```

2. Dereferenziazione di iteratori invalidi:

```
auto it = v.begin();
v.clear();  // Invalida tutti gli iteratori
*it;  // Errore: undefined behavior!

// Pattern corretto:
for(auto it = v.begin(); it != v.end(); ) {
    if(condition) {
        it = v.erase(it);  // erase() ritorna iteratore valido
    } else {
        ++it;
    }
}
```

3. Reference a oggetti temporanei:

```
T& ref = T();  // Errore: reference a temporaneo
const T& ref = T();  // OK: estende vita del temporaneo

// Pattern corretto:
T obj;
T& ref = obj;
```

Pattern di Conversione dei Tipi

Q: Come gestire i vari tipi di cast all'interno dei cicli?

A:

1. Cast interno al ciclo (quando serve verificare ogni elemento):

```
for(vector<Base*>::iterator it = v.begin(); it != v.end(); ++it) {
    // Cast diretto per verifica
    if(Derived* d = dynamic_cast<Derived*>(*it)) {
        // Operazioni su d
    }
}
```

2. Cast con const (quando serve mantenere la costness):

```
// Da puntatore non-const a puntatore const
const Derived* d = dynamic_cast<const Derived*>(base_ptr);

// Con iteratori const
for(vector<const Base*>::const_iterator cit = v.cbegin(); cit != v.cend();
++cit) {
    if(const Derived* d = dynamic_cast<const Derived*>(*cit)) {
        // Operazioni read-only su d
    }
}
```

3. Cast con rimozione const (quando necessario modificare):

```
// Rimozione const e cast
Derived* d = dynamic_cast<Derived*>(const_cast<Base*>(const_base_ptr));

// Esempio da gerarchia IOS:
istream* i = dynamic_cast<istream*>(const_cast<ios*>(&stream));
```

Q: Quali sono i pattern comuni per gestire dynamic_cast in cicli?

A:

1. Pattern per controllo tipo e operazione:

```
// Esempio con gerarchia QT
for(auto it = widgets.begin(); it != widgets.end(); ++it) {
    if(QButton* btn = dynamic_cast<QButton*>(*it)) {
        btn->setText("Label");
        result.push_back(btn);
    }
}
```

2. Pattern per verifiche multiple:

```
for(auto* component : components) {
    // Verifica singola
    if(Button* btn = dynamic_cast<Button*>(component)) {
        buttons.push_back(btn);
    }

    // Verifiche multiple (OR logico)
    if(dynamic_cast<QCheckBox*>(*it) || dynamic_cast<QPushButton*>(*it)) {
        // Operazioni
    }
}
```

3. Pattern con operazioni condizionali:

```
for(auto cit = v.begin(); cit != v.end(); ++cit) {
    // Cast multipli per verifica condizioni
    istream* i = dynamic_cast<istream*>(const_cast<ios*>(*cit));
    ostream* o = dynamic_cast<ostream*>(const_cast<ios*>(*cit));

    if(i && o) {  // Verifica entrambi i cast
        if(i->tellg() > o->tellp()) {
            i->setstate(ios::goodbit);
        }
    }
}
```

Q: Come gestire la memoria durante le conversioni?

A:

1. Pattern di allocazione sicura:

```
Button** buttonsArray = new Button*[buttons.size()];

for(size_t i = 0; i < buttons.size(); i++) {
        buttonsArray[i] = dynamic_cast<Button*>(components[i]);
        if(!buttonsArray[i]) {
            delete[] buttonsArray;  // Cleanup in caso di errore
            throw Fallimento(); // classe di eccezzioni creata ad hoc
        }
}
```

2. Pattern di deallocazione con cast:

```
for(auto it = list.begin(); it != list.end(); ) {
    if(auto* ptr = dynamic_cast<Derived*>(*it)) {
        delete ptr;  // Dealloca oggetto
        it = list.erase(it);  // Rimuove puntatore
    } else {
        ++it;
    }
}
```

Classe di eccezione ad hoc

Due alternative:

1. Creo una classe vuota

```
class Fallimento{};

...

if(..) throw Fallimento();
```

3. Creo una classe con un costruttore

```
class Fallimento{
        private:
                std::string msg;
        public:
                Fallimento(std::string m): msg(m) {};
};

...

if(..) throw Fallimento("MSG");
```

Best Practices Generali

1. **Controlli Null**:
    - Sempre verificare puntatori prima dell'uso
    - Usare nullptr invece di NULL
2. **Gestione Memoria**:
    - RAII quando possibile

    - Smart pointers per casi complessi
    - Delete esplicito quando necessario
3. **Iterazione**:
    - Preferire range-based for quando possibile
    - Gestire correttamente iteratori durante modifiche
4. **Const Correctness**:
    - Marcare const tutto ciò che non deve essere modificato
    - Usare const_iterator per iterazione read-only