18 - Smart Pointers

La gestione manuale della memoria con i puntatori raw (detti anche "naked") può introdurre errori come:

- *memory leak*
- *dangling pointer*
- *double delete*
  Per risolvere questi problemi, si usa l'idioma **RAII (Resource Acquisition Is Inizialization)**. Questo approccio prevede che una risorsa venga gestita da un oggetto il cui costruttore la acquisisce e il cui distruttore la rilascia automaticamente.
  C++ offre tre classi principali di puntatori smart nella libreria `<memory>`:

1. `std::unique_ptr`: un puntatore che ha unicità di proprietà (owning).
2. `std::shared_ptr`: un puntatore che permette la condivisione della proprietà.
3. `std::weak_ptr`: un puntatore che non partecipa alla gestione attiva della risorsa.

1 - `std::unique_ptr`:

- gestisce una risorsa in modo esclusivo;
- non è copiabile, ma è spostabile (movable);
- rilascia automaticamente la risorsa quando esce dallo scope.

```
#include <memory>

void foo() {
    std::unique_ptr<int> pi(new int(42)); // Gestisce un intero
    *pi = 100;                            // Dereferenziazione come un normale
puntatore
} // Alla fine di foo(), pi rilascia la memoria.
```

Caratteristiche principali:

- **non copiabile**: evita duplicazioni che potrebbero portare a doppio rilascio della risorsa;
- **spostamento**: con `std::move`, la proprietà della risorsa può essere trasferita:
  `cpp void foo(std::unique_ptr<int> p); std::unique_ptr<int> ptr(new int(42)); foo(std::move(ptr)); // Trasferisce la proprietà della risorsa`
  Metodi importanti:
- `reset(raw_ptr)`: cambia la risorsa gestita, rilasciando eventualmente quella precedente;
- `get()`: restituisce il puntatore raw (la gestione resta allo `unique_ptr`);
- `release()`: rimuove le proprietà della risorsa e restituisce il puntatore raw (ora il programmatore deve gestire la memoria manualmente).
  Vantaggi:
- leggero ed efficiente;
- adatto quando è garantita l'unicità della proprietà.

2 - `std::shared_ptr`:

- gestisce una risorsa in modo condiviso tra più puntatori;
- usa un reference counter per tracciare quante copie dello shared pointer esistono;
- rilascia la risorsa solo quando il reference counter scende a 0.

```
#include <memory>

void foo() {
    std::shared_ptr<int> sp1(new int(42)); // Ref counter = 1
    {
        std::shared_ptr<int> sp2 = sp1;    // Ref counter = 2
        *sp2 = 50;                         // Modifica condivisa
    } // Ref counter = 1 (sp2 esce dallo scope)
} // Ref counter = 0, la risorsa viene rilasciata.
```

Caratteristiche principali:
copiabile e spostabile: la copia aumenta il reference counter, lo spostamento no.

```
void foo(std::shared_ptr<int> sp);
std::shared_ptr<int> sp(new int(42));
foo(sp);             // Copia condivisa, ref counter aumenta
foo(std::move(sp));  // Spostamento, ref counter non cambia
```

Metodi importanti:

- `reset()` : rilascia la risorsa e può assegnarne una nuova;
- `get()` : restituisce il puntatore raw (senza trasferire la proprietà).
- `std::make_shared` : consente di creare un `shared_ptr` in modo efficiente, allocando risorsa e blocco di controllo in un'unica operazione.

```
auto sp = std::make_shared<int>(42); // Più efficiente di `new`
```

Vantaggi:

- ideale per scenari in cui più entità devono accedere alla stessa risorsa;
- la risorsa viene rilasciata automaticamente.

3 - `std::weak_ptr` :

- è un puntatore smart che punta a una risorsa gestita da uno `shared_ptr` , senza incrementare il reference counter;
- utile per evitare cicli di riferimento (es. in strutture dati come grafi o alberi).

```
#include <memory>
#include <iostream>

void maybe_print(std::weak_ptr<int> wp) {
    if (auto sp = wp.lock()) { // Converte in shared_ptr se la risorsa è disponibile
        std::cout << *sp;
    } else {
        std::cout << "Risorsa non più disponibile";
```

```
    }
}

void foo() {
    std::weak_ptr<int> wp;
    {
        auto sp = std::make_shared<int>(42);
        wp = sp; // wp osserva sp
        maybe_print(wp); // Stampa: 42
    } // sp viene distrutto, la risorsa non è più disponibile
    maybe_print(wp); // Stampa: Risorsa non più disponibile
}
```

Vantaggi:

- risolve problemi di cicli di riferimento;
- evita memory leak in strutture con dipendenze reciproche.

Funzioni `std::make_shared` e `std::make_unique`

Le funzioni `std::make_shared` e `std::make_unique` consentono di creare puntatori smart senza usare esplicitamente `new`. Questo approccio:

- è più efficiente;
- previene problemi di exception safety.

```
void foo() {
    //codice NON exception safe
    bar(std::shared_ptr<int>(new int(42)), std::shared_ptr<int>(new int(42)));

    //codice exception safe
    bar(std::make_shared<int>(42), std::make_shared<int>(42));
}
```

nella prima chiamata di `bar` l'implementazione potrebbe decidere di valutare:

- prima le due `new int(42)` che sono argomenti dei costruttori dei due `shared_ptr`;
- solo dopo invocare i costruttori dei due `shared_ptr`.
  se la prima allocazione va a buon fine ma la seconda invece fallisce con eccezione, si ottiene un memory leak per la prima risorsa allocata.
  Il problema non si presenta nella seconda chiamata a `bar`, perché le allocazioni sono effettuate (implicitamente) dalla `make_shared`.

Casi particolari e linee guida

1. **cicli di riferimento**:
   - usare `std::weak_ptr` quando si creano dipendenze circolari tra shared pointers.
2. **evita `new` e `delete` diretti**:
   - le linee guida moderne del C++ sconsigliano l'uso di `new` e `delete`, privilegiando gli smart pointer.
3. **performance**:

- usare `std::unique_ptr` dove possibile, perché è più leggero rispetto a `std::shared_ptr`.

Appello_20120202 es. 5
La classe seguente contiene errori inerenti la corretta gestione delle risorse. Individuare almeno due problemi logicamente distinti, indicando la sequenza di operazioni che porta alla loro occorrenza. Fornire quindi una soluzione alternativa e discutere brevemente i motivi per i quali tale soluzione si può ritenere corretta.

```
#include <string>
class A {
    int* pi; //puntatore ad intero
    std::string str;
    double* pd; //puntatore a double
public:
    A(const std::string& s) : pi(new int), str(s), pd(new double) { } //costruttore
    ~A() { delete pi; delete pd; } //distruttore
};
```

risoluzione:
due problemi logicamente distinti:

1. non posso considerare il distruttore e il costruttore senza tenere presente anche il costruttore di copia e l'operatore di assegnamento (senza, gli oggetti vengono copiati con una shallow copy (vengono copiati gli *indirizzi* di pi e pd) e troviamo due oggetti A che puntano allo stesso intero e allo stesso double, se uno dei due finisce il ciclo di vita rilasciando le risorse lascerà l'altro con dei dangling pointers, se prova anche a distruggerle farà una double free);
2. se nel costruttore viene lanciata un'eccezione e l'oggetto non è ancora stato creato "completamente" la memoria allocata fino a quel momento verrà persa (memory leak) in quanto non è possibile chiamare il distruttore.

```
//modalità #1: smart pointers
#include <string>
class A {
    std::unique_ptr<int> pi; //non è possibile fare la copia
    std::string str;
    std::unique_ptr<double> pd;
public:
    A(const std::string& s) : pi(new int), str(s), pd(new double) { }

    A(const A&) = delete;
    A& operator=(const A&) = delete; //disabilito la copia

    A(A&&) = default;
    A& operator=(A&&) = default; //spostamento

    ~A() = default;
};

//modalità #2: try-catch
```

```cpp
#include <string>
class A {
    int* pi;
    std::string str;
    double* pd;
public:
    A(const std::string& s)
        : pi(nullptr), str(s), pd(nullptr) {
        pi = new int;
        try {
            pd = new double;
        } catch (...) {
            delete pi;
            throw;
        }
    }
    ~A() { //non può lanciare eccezioni
        delete pd;
        delete pi;
    }
};
```

Appello_20220907 es. 3

Il seguente codice non ha un comportamento corretto in presenza di eccezioni. Individuare almeno un problema, indicando la sequenza di operazioni che porta alla sua occorrenza. Fornire quindi una soluzione basata sull'utilizzo dei blocchi try/catch.

```cpp
void load_and_process(const std::string& conn_params,
                      const std::string& query) {
    Connection conn;
    Results res;

    conn.open(conn_params); //acquisizione connessione

    res.init(); //acquisizone buffer per risultati
    conn.execute(query, res); //caricamento dati
    process(res); //elaborazione dati
    res.finish(); //rilascio buffer -- non lancia eccezioni (distruzione esplicita,
non è applicato l'idioma RAII)
    conn.close(); //rilascio connessiome -- non lancia eccezioni    (distruzione
esplicita, non è applicato l'idioma RAII)
}
```

risoluzione:

```cpp
void load_and_process(const std::string& conn_params,
                      const std::string& query) {
    Connection conn;
    Results res;

    conn.open(conn_params); // se l'acquisizione dà errore (quindi non viene
```

```
acquisita la risorsa) non c'è bisogno di liberarla
    try {
        res.init(); //acquisizione
        try {
            conn.execute(query, res);
            process(res);
            res.finish();
            conn.close();
        } catch (...) {
            res.finish();
            throw;
        }
    } catch (...) {
        conn.close();
        throw;
    }
}
```