Cosa Stampa: Casistiche Complete ed Errori

Metodologia di Risoluzione

Track Sistematico

1. **Identificare tipo statico (TS) e tipo dinamico (TD)**
2. **Verificare presenza di** `virtual` → se sì, vai al tipo dinamico
3. **Controllare signature esatte** → override richiede identità completa
4. **Seguire catena di ereditarietà** → dal TD verso TS se necessario
5. **Gestire conversioni e cast** → analizzare effetti sui tipi

Casistiche Critiche di Errore

❌ CASO 1: Signature Mismatch (Override Fallito)

cpp

```cpp
class A {
    virtual void g() const { cout << "A::g"; }  // const!
};
class C : public A {
    virtual void g() { cout << "C::g"; }        // NO const!
};

A* p3 = new C();
static_cast<A*>(p3->n())->g();  // → "A::g" (NON C::g!)
```

ERRORE: `C::g()` NON fa override di `A::g() const` **MOTIVO**: Signature diverse (`const` vs non-`const`) **CONSEGUENZA**: Viene chiamato `A::g()` sempre

❌ CASO 2: Problemi con `return this`

cpp

```cpp
class A {
    virtual A* n() { cout << "A::n"; return this; }
    virtual void g() const { cout << "A::g"; }
};
class C : public A {
    // Non ridefinisce g() in modo compatibile
};

// Conversione avviene su *this!
(static_cast<A*>(p3->n()))->g();
```

PROBLEMA: La conversione su `*this` "riporta sempre alla classe base" **REGOLA**: Quando si converte il risultato di `return this`, il comportamento successivo parte dalla classe base **MORALE**: Tutte le stampe su `*this` convertito restano nella classe base

❌ CASO 3: Metodi Non-Virtual dopo Conversione

cpp

```
(p3->n())->m();  // m() NON è virtual
```

SCENARIO:

- `p3->n()` chiama metodo virtual → stampa corretta del tipo dinamico
- `return this` riporta alla classe base
- `m()` non-virtual → usa implementazione classe base

OUTPUT: `A::n A::m A::g C::j` (mix di chiamate)

❌ CASO 4: Ricorsione Infinita

cpp

```
class C : public A {
    void k() const { cout << "C::k"; k(); }  // RICORSIONE!
};

static_cast<C*>(p2)->k();  // → Stack overflow
```

ERRORE: `k()` chiama se stesso indefinitamente **DIAGNOSI**: Se non presente opzione UB, diventa errore runtime **PREVENZIONE**: Controllare sempre chiamate ricorsive

❌ CASO 5: Problemi di Const-Correctness

cpp

```
const A* p5 = new C();
(p5->n())->g();  // NC (Non Compila)
```

ERRORE: Puntatore `const` ritorna `const A*`, ma `g()` non è `const` **REGOLA**: `const` object può chiamare solo metodi `const` **SIMMETRICO**: Metodo `const` che ritorna non-`const` pointer → errore

Casi di Non Compilazione (NC)

NC 1: Const Object → Non-Const Method

cpp

```cpp
const B* ptr;
ptr->non_const_method();  // ERRORE
```

NC 2: Const Method → Non-Const Return

cpp

```cpp
virtual const B* j() {return *this};  // ERRORE se chiamato su non-const
```

NC 3: Cross-Cast Invalido

cpp

```cpp
static_cast<UnrelatedClass*>(ptr);  // ERRORE se non c'è relazione
```

Casi di Ereditarietà Multipla

Diamond Problem

cpp

```cpp
class F : public B, public E, public D  // Ordine importante!
```

COSTRUZIONE: Da sinistra → B() C() D() E() F() **DISTRUZIONE**: Da destra → ~F() ~E() ~D() ~C() ~B()

Method Resolution

cpp

```cpp
dynamic_cast<C*>(pa3)->f(Z(2));  // → F::f(Z)
```

REGOLA: Prende il metodo "più vicino" al tipo effettivo dell'oggetto **ALGORITMO**: Cerca nella gerarchia dal tipo dinamico verso l'alto

Undefined Behavior (UB)

Quando si Verifica

1. **Ricorsione infinita** senza controllo
2. **Access a membri dopo delete**
3. **Cast invalidi** con static_cast
4. **Virtual call durante costruzione/distruzione**

Gestione negli Esercizi

- **Con UB**: Comportamento imprevedibile
- **Senza UB**: Errore runtime o stack overflow

Regole di Costruzione/Distruzione

Costruzione

cpp

```cpp
B* p1 = new F();  // B() C() D() E() F()
```

- **Ordine**: Dalla classe base alle derivate
- **Ereditarietà multipla**: Da sinistra a destra
- **Tipo**: Usa tipo dinamico (F), non statico (B)

Distruzione

cpp

```cpp
delete p1;  // ~F() ~E() ~D() ~C() ~B()
```

- **Ordine**: Contrario alla costruzione
- **Ereditarietà multipla**: Da destra a sinistra
- **Virtual destructor**: Necessario per comportamento corretto

Pattern di Verifica Tipo

typeid per Uguaglianza Esatta

cpp

```cpp
if (typeid(*ptr) == typeid(ConcreteClass)) {
    // ptr punta ESATTAMENTE a ConcreteClass
}
```

dynamic_cast per Compatibilità

cpp

```cpp
if (auto* derived = dynamic_cast<DerivedClass*>(ptr)) {
    // ptr è compatibile con DerivedClass
}
```

Debugging Checklist

Prima di Analizzare

- [ ] Identificare gerarchia di ereditarietà completa
- [ ] Verificare signature ESATTE di tutti i metodi virtual
- [ ] Controllare presenza/assenza di `const` qualifiers
- [ ] Mappare tutti i `return this` e loro effetti

Durante l'Analisi

- [ ] Seguire tipo statico vs dinamico passo per passo
- [ ] Verificare ogni conversione di cast
- [ ] Controllare se i metodi sono virtual o meno
- [ ] Tracciare modifiche al tipo statico

Casi Edge da Verificare

- [ ] Method hiding vs method overriding
- [ ] Const-correctness in tutta la catena
- [ ] Ereditarietà multipla e ordine di risoluzione
- [ ] Presenza di undefined behavior

Errori Comuni negli Esami

1. **Confondere hiding con overriding**
2. **Ignorare const qualifiers nelle signature**
3. **Non considerare effetto di return this su casting**
4. **Assumere che dynamic_cast funzioni sempre**
5. **Non tracciare modifiche al tipo statico durante catene di chiamate**

Strategia di Risoluzione Rapida

1. **Disegna la gerarchia** con tutti i metodi e signature
2. **Traccia ogni chiamata** annotando TS/TD
3. **Evidenzia ogni cast** e suo effetto
4. **Verifica const-correctness** passo per passo
5. **Controlla ricorsioni** e chiamate infinite
6. **Applica regole costruzione/distruzione** se presenti

REGOLA D'ORO: Quando in dubbio, il compilatore C++ è estremamente rigoroso con le signature e la const-correctness. Un singolo qualificatore diverso cambia completamente il comportamento.