Static_cast vs Dynamic_cast: Analisi Formale Completa

Definizioni Teoriche

static_cast<T>(expr)

Natura: Conversione verificata al **compile-time Filosofia**: "Fidati del programmatore" **Controlli**: Zero verifiche runtime sulla validità semantica **Overhead**: O(1) - nessun costo computazionale **Fallimento**: Non può fallire tecnicamente (al massimo undefined behavior)

dynamic_cast<T>(expr)

Natura: Conversione verificata al **runtime** tramite RTTI **Filosofia**: "Verifica sempre la sicurezza" **Controlli**: Completa validazione della gerarchia di ereditarietà **Overhead**: O(h) dove h = altezza gerarchia (traversal vtable) **Fallimento**: Ritorna `nullptr` (puntatori) o lancia `std::bad_cast` (riferimenti)

Regole Operative Formali

Precondizioni static_cast

1. **Compatibilità sintattica**: Deve esistere un percorso di conversione implicita/esplicita
2. **Responsabilità programmatore**: Garanzia semantica delegata completamente al dev
3. **Tipo polimorfo**: NON richiesto (funziona anche con tipi non-polimorfi)

Precondizioni dynamic_cast

1. **Tipo polimorfo obbligatorio**: Classe sorgente deve avere ≥1 funzione virtuale
2. **Gerarchia valida**: Tipo target deve essere nella gerarchia del tipo sorgente
3. **RTTI abilitato**: Compilazione con informazioni di tipo runtime

Algoritmo Decisionale Rigoroso

cpp

```cpp
// Matrice decisionale
if (conversione_built_in || upcasting_garantito_staticamente) {
    → static_cast
}
else if (downcast_incerto && robustezza_richiesta) {
    → dynamic_cast + controllo_nullptr
}
else if (performance_critica && garanzia_algoritmica_preesistente) {
    → static_cast + assert(typeid(*ptr) == typeid(Target))
}
else {
    → dynamic_cast  // Default sicuro
}
```

Casistiche Critiche

Caso 1: Conversione su `return this`

cpp

```cpp
// PROBLEMA: static_cast con return this
class A {
    virtual A* n() { return this; }
    virtual void g() const { /* */ }
};
class C : public A {
    virtual void g() { /* signature diversa! */ }
};

A* p3 = new C();
static_cast<A*>(p3->n())->g();  // Chiama A::g(), NON C::g()
```

Spiegazione:

- `p3->n()` ritorna `this` di tipo `C*`
- `static_cast<A*>` converte forzatamente a `A*`
- Poiché `C::g()` ha signature diversa da `A::g() const`, non c'è override
- Viene chiamato `A::g() const`

Caso 2: dynamic_cast fallisce

cpp

```cpp
dynamic_cast<B*>(p3->n())->g();  // → nullptr
```

Motivo: Il `dynamic_cast` verifica runtime e fallisce perché la conversione non è semanticamente valida nella gerarchia effettiva.

Pattern di Implementazione

Pattern Sicuro (dynamic_cast)

cpp

```cpp
if (auto* target = dynamic_cast<Derived*>(base_ptr)) {
    target->specific_method();  // Tipo garantito
} else {
    // Gestione fallimento controllata
}
```

Pattern Ottimizzato (static_cast)

cpp

```cpp
// Solo dopo verifica esplicita
assert(typeid(*base_ptr) == typeid(Derived));
auto* target = static_cast<Derived*>(base_ptr);
target->specific_method();  // Performance massima
```

Regole di Scelta Strategica

Usa static_cast quando:

1. **Upcasting garantito**: `Derived*` → `Base*`
2. **Conversioni primitive**: `int` → `double`
3. **Performance critica**: Hot paths con garanzie algoritmiche
4. **Controllo manuale**: Hai già verificato il tipo tramite logica applicativa

Usa dynamic_cast quando:

1. **Downcast incerto**: `Base*` → `Derived*` senza certezze
2. **Cross-cast**: Navigazione ereditarietà multipla
3. **Robustezza**: Fallimento deve essere gestibile gracefully
4. **API pubbliche**: Dove non controlli l'input

Errori Comuni e Debugging

Errore 1: Assumere override quando non c'è

cpp

```cpp
// SBAGLIATO: signature diverse
virtual void method() const;  // Base
virtual void method();        // Derived - NON È OVERRIDE!
```

Errore 2: static_cast su gerarchie complesse

cpp

```cpp
// PERICOLOSO senza verifica
static_cast<SiblingClass*>(ptr);  // Undefined behavior se sbagliato
```

Errore 3: dynamic_cast senza controllo nullptr

cpp

```cpp
// CRASH POTENZIALE
dynamic_cast<Derived*>(ptr)->method();  // Se nullptr → segfault
```

Verifiche Runtime

Con typeid (verifica esatta)

cpp

```cpp
if (typeid(*ptr) == typeid(ConcreteType)) {
    // Tipo dinamico è ESATTAMENTE ConcreteType
}
```

Con dynamic_cast (verifica compatibilità)

cpp

```cpp
if (dynamic_cast<BaseType*>(ptr)) {
    // ptr è compatibile con BaseType (o suoi sottotipi)
}
```

Performance Considerations

Operazione	static_cast	dynamic_cast
Compile-time	✅ Risolto	❌ Info RTTI
Runtime	O(1)	O(log h)
Memory	Zero overhead	RTTI tables
Debug	Difficile	Intrinseco

Linee Guida Architetturali

1. **Default a dynamic_cast** per sicurezza
2. **Profiling-driven optimization** a static_cast solo dopo misurazione
3. **Documentazione esplicita** delle assunzioni nei static_cast
4. **Unit testing** intensivo per tutti i cast non-triviali
5. **Static analysis tools** per rilevare cast pericolosi