🎯 GUIDA COMPLETA ESAME PROGRAMMAZIONE AD OGGETTI

📋 INDICE CATEGORIE

---

1. COSA STAMPA

🔍 METODOLOGIA STEP-BY-STEP

```
STEP 1: Disegna la gerarchia con TUTTI i metodi e signature esatte
STEP 2: Per ogni chiamata, identifica Tipo Statico (TS) e Tipo Dinamico (TD)
STEP 3: Applica regole di risoluzione metodi
STEP 4: Traccia ogni cast e suo effetto sui tipi
STEP 5: Verifica const-correctness
```

⚙️ REGOLE DI RISOLUZIONE

Condizione	Azione
Metodo virtual	Vai al tipo dinamico (TD)
Metodo non-virtual	Usa tipo statico (TS)
Override presente	Chiama versione del TD
Override assente	Risali gerarchia verso TS
Signature diverse	NON è override!

🚨 TRAPPOLE COMUNI

🚨 TRAPPOLA 1: Signature Mismatch

```
class A { virtual void g() const; };      // const!
class C { virtual void g(); };            // NO const!
// → NON è override, sempre A::g()
```

🚨 TRAPPOLA 2: Return This + Cast

```
static_cast<A*>(p3->n())->g();
// 1. p3->n() → TD corretto
// 2. cast → TS diventa A*
// 3. g() → parte da A, non da TD originale
```

🚨 TRAPPOLA 3: Const-Correctness

```
const A* ptr;
ptr->non_const_method();  // NC (Non Compila)
```

✅ CHECKLIST COSA STAMPA

- ☐ Gerarchia disegnata con signature complete
- ☐ TS/TD tracciati per ogni chiamata
- ☐ Cast verificati e effetti annotati
- ☐ Const-correctness controllata
- ☐ Ricorsioni infinite identificate
- ☐ Override vs hiding verificato

---

2. SOTTOTIPI

🎯 METODOLOGIA STEP-BY-STEP

```
STEP 1: Leggi il codice della funzione template
STEP 2: Identifica cosa fanno le chiamate (cout stampe specifiche)
STEP 3: Deduce le relazioni dalla logica delle chiamate
STEP 4: Verifica che tutte le conversioni compilino
STEP 5: Costruisci gerarchia compatibile
```

⚙️ REGOLE DI DEDUZIONE

```
template <class X, class Y>
X* Fun(X* p) { return dynamic_cast<Y*>(p); }

// Se fun<A,B>(new C()) stampa "Alan" → B è sottotipo di A
// Se dynamic_cast<C*>(new B()) == 0 → C NON è sottotipo di B
```

🔍 ALGORITMO RISOLUZIONE

1. **Identifica stampe certe** (es. "Alan", "Turing")
2. **Mappa stampe → condizioni** (es. "Alan" se cast riuscito)
3. **Deduce relazioni** da successi/fallimenti cast
4. **Verifica coerenza** con tutte le chiamate
5. **Segna VERO/FALSO/POSSIBILE** per ogni relazione

📝 TEMPLATE RISPOSTA

```
A<=B: FALSO   (A non è sottotipo di B)
A<=C: FALSO
A<=D: FALSO
B<=A: VERO    (B è sottotipo di A)
B<=C: FALSO
B<=D: FALSO
C<=A: VERO
C<=B: FALSO
C<=D: POSSIBILE  (compatibile ma non determinabile)
D<=A: VERO
D<=B: VERO
D<=C: POSSIBILE
```

---

3. FUNZIONI

🎯 METODOLOGIA STEP-BY-STEP

```
STEP 1: Analizza signature richiesta e parametri
STEP 2: Identifica operazioni necessarie (filtro, trasformazione, conteggio)
STEP 3: Scegli struttura dati (vector<T*> di solito)
STEP 4: Implementa ciclo con iteratori appropriati
STEP 5: Usa typeid/dynamic_cast per controlli tipo
```

⚙ TEMPLATE FUNZIONI LETTURA (const)

```
// Per funzioni che NON modificano il container
ReturnType functionName(const vector<BaseType*>& container) {
    ReturnType result;

    for (auto it = container.cbegin(); it != container.cend(); ++it) {
        // *it è BaseType* const (puntatore const)
        // **it è const BaseType& (oggetto const)

        // Controllo tipo esatto
        if (typeid(**it) == typeid(ConcreteType)) {
            // Cast a tipo costante
            const auto* concrete = dynamic_cast<const ConcreteType*>(*it);
            result += concrete->constMethod();
        }
    }
    return result;
}
```

⚙ TEMPLATE FUNZIONI MODIFICA (non-const)

```
// Per funzioni che MODIFICANO il container
void functionName(vector<BaseType*>& container) {

    // RIMOZIONE con erase-remove idiom
    container.erase(
        remove_if(container.begin(), container.end(),
            [](BaseType* ptr) {
                if (typeid(*ptr) == typeid(TypeToRemove)) {
                    delete ptr;  // PRIMA delete
                    return true; // POI rimuovi dal container
                }
                return false;
            }),
        container.end()
    );

    // ALTERNATIVA: Rimozione manuale con iteratori
    for (auto it = container.begin(); it != container.end();) {
        if (/* condizione rimozione */) {
            delete *it;           // Delete oggetto
            it = container.erase(it);  // Rimuovi e aggiorna iteratore
        } else {
            ++it;                 // Avanza solo se non rimosso
        }
    }
}
```

⚙ GESTIONE PUNTATORI: * vs & vs const_cast

```
void exampleFunction(vector<BaseType*>& container) {
    for (auto it = container.begin(); it != container.end(); ++it) {

        // *it → BaseType* (puntatore all'oggetto)
        BaseType* ptr = *it;

        // **it → BaseType& (riferimento all'oggetto)
        BaseType& obj = **it;

        // QUANDO USARE const_cast
        if (/* container è const ma devo modificare oggetto */) {
            auto* mutable_ptr = const_cast<BaseType*>(*it);
            mutable_ptr->nonConstMethod();
        }

        // QUANDO USARE dynamic_cast const
        if (const auto* derived = dynamic_cast<const DerivedType*>(*it)) {
            // Accesso solo lettura al tipo derivato
            derived->constMethod();
        }

        // QUANDO USARE dynamic_cast non-const
        if (auto* derived = dynamic_cast<DerivedType*>(*it)) {
            // Accesso modifica al tipo derivato
            derived->nonConstMethod();
        }
    }
}
```

□ SCELTE IMPLEMENTATIVE

Scenario	Strumento	Esempio
"Tipo esattamente X"	`typeid(*ptr) == typeid(X)`	Conteggio tipo specifico
"È sottotipo di X"	`dynamic_cast<X*>(ptr) != nullptr`	Operazioni su gerarchia
"Conteggio/somma"	Contatore o accumulatore	`count++` o `sum += value`
"Filtraggio"	Nuovo container + controlli tipo	Copia selettiva
"Rimozione"	`erase` + `delete`	Pulizia container
"Trasformazione"	Modifica in-place o nuovo container	Modifica oggetti esistenti

□ REGOLE CRITICHE MEMORY MANAGEMENT

```
// ❌ ERRORE COMUNE: Memory leak
container.erase(it);  // Rimuove dal container ma NON dealloca memoria

// ✅ CORRETTO: Prima delete, poi erase
delete *it;           // Dealloca memoria
it = container.erase(it);  // Rimuove dal container

// ✅ PATTERN SICURO con remove_if
container.erase(
    remove_if(container.begin(), container.end(),
        [](BaseType* ptr) {
            if (/* condizione */) {
                delete ptr;    // Prima dealloca
                return true;   // Poi marca per rimozione
            }
            return false;
        }),
    container.end()
);
```

🔍 DECISION TREE: const vs non-const

```
Devo modificare il container?
├─ SÌ → vector<T*>& + iterator + delete/erase se necessario
└─ NO → const vector<T*>& + const_iterator + dynamic_cast<const T*>

Devo modificare gli oggetti nel container?
├─ SÌ → dynamic_cast<T*> + metodi non-const
└─ NO → dynamic_cast<const T*> + metodi const

Container const ma oggetti modificabili?
└─ const_cast<T*> (SOLO se semanticamente corretto)
```

✅ CHECKLIST FUNZIONI

- [ ] Signature corretta (const-correctness)
- [ ] Iteratori appropriati (const_iterator se necessario)
- [ ] Controlli tipo corretti (typeid vs dynamic_cast)
- [ ] Gestione return value
- [ ] Nessun memory leak (se allochi)

---

4. MODELLAZIONI

🎯 METODOLOGIA STEP-BY-STEP

```
STEP 1: Leggi attentamente le specifiche del dominio
STEP 2: Identifica classi base e derivate
STEP 3: Determina metodi comuni (→ classe base)
STEP 4: Progetta gerarchia con virtual appropriati
STEP 5: Implementa parte 2 (container + funzioni)
```

⚙️ TEMPLATE CLASSE BASE

```
class BaseClass {
private:
    // Campi comuni

protected:
    // Costruttori (se derivazione)
    BaseClass(/* parametri */);

public:
    // Costruttore pubblico
    BaseClass(/* parametri */);

    // Distruttore virtuale (OBBLIGATORIO per polimorfismo)
    virtual ~BaseClass() = default;

    // Getters per campi privati
    ReturnType getField() const { return field; }

    // Metodi virtuali puri (se classe astratta)
    virtual ReturnType pureMethod() const = 0;

    // Metodi virtuali con implementazione default
    virtual ReturnType method() const { /* implementazione */ }
};
```

⚙ TEMPLATE CLASSE DERIVATA

```
class DerivedClass : public BaseClass {
private:
    // Campi specifici

public:
    // Costruttore
    DerivedClass(/* parametri */) : BaseClass(/* parametri base */), /* init specifici */ {}

    // Override metodi virtuali puri
    ReturnType pureMethod() const override { /* implementazione */ }

    // Override opzionali
    ReturnType method() const override { /* implementazione specifica */ }

    // Metodi specifici
    void specificMethod() const { /* implementazione */ }
};
```

PARTE 2: CONTAINER + FUNZIONI

```
class ContainerClass {
private:
    vector<BaseClass*> items;

public:
    // Costruttore
    ContainerClass() = default;

    // Distruttore (pulisce memoria)
    ~ContainerClass() {
        for (auto* item : items) {
            delete item;
        }
    }

    // Metodi richiesti
    void addItem(BaseClass* item) { items.push_back(item); }

    // Implementa funzioni richieste usando template da sezione 3
    ReturnType requestedFunction() const {
        // Usa template funzioni
    }
};
```

✅ CHECKLIST MODELLAZIONI

- ☐ Gerarchia progettata correttamente
- ☐ Distruttore virtuale nella base
- ☐ Metodi virtuali puri se richiesti
- ☐ Costruttori appropriati
- ☐ Getters per campi privati
- ☐ Container con vector<Base*>
- ☐ Gestione memoria nel distruttore
- ☐ Funzioni implementate correttamente

---

5. DEFINIZIONI STANDARD

🎯 METODOLOGIA STEP-BY-STEP

```
STEP 1: Identifica da quali classi deriva direttamente
STEP 2: Scegli tipo (costruttore copia vs operatore assegnazione)
STEP 3: Applica template appropriato
STEP 4: Richiama costruttori/operatori delle classi base
STEP 5: Assegna campi specifici della classe
```

⚙️ TEMPLATE COSTRUTTORE DI COPIA

```
// Per classe con ereditarietà singola
ClassName(const ClassName& other)
    : BaseClass(other),           // Chiama costruttore copia base
      field1(other.field1),       // Copia campi specifici
      field2(other.field2) {
    // Eventuali operazioni aggiuntive
}

// Per ereditarietà multipla
ClassName(const ClassName& other)
    : Base1(other),               // Prima base
      Base2(other),               // Seconda base
      field1(other.field1) {
    // Operazioni aggiuntive
}
```

⚙ TEMPLATE OPERATORE ASSEGNAZIONE

```
ClassName& operator=(const ClassName& other) {
    if (this != &other) {          // Controllo auto-assegnazione
        BaseClass::operator=(other); // Chiama operatore base
        // o Base1::operator=(other); Base2::operator=(other); per multipla

        field1 = other.field1;     // Assegna campi specifici
        field2 = other.field2;
    }
    return *this;
}
```

□ REGOLE SPECIFICHE

Scenario	Azione
Ereditarietà singola	`BaseClass(other)`
Ereditarietà multipla	`Base1(other), Base2(other)`
Campi puntatore	Deep copy se necessario
Const fields	Solo costruttore copia (no assegnazione)

□ CHECKLIST DEFINIZIONI STANDARD

- ☐ Identificate tutte le classi base dirette
- ☐ Chiamate costruttori/operatori delle basi
- ☐ Copiati tutti i campi specifici
- ☐ Controllo auto-assegnazione (operatore=)
- ☐ Return *this (operatore=)
- ☐ Gestiti campi puntatore se presenti

---

6. GERARCHIE

□ METODOLOGIA STEP-BY-STEP

```
STEP 1: Leggi ogni specifico requisito della classe
STEP 2: Determina quali classi sono astratte/concrete
STEP 3: Identifica metodi virtuali puri necessari
STEP 4: Progetta costruttori (public/protected)
STEP 5: Implementa definizioni standard se richieste
```

⚙ CLASSIFICAZIONE CLASSI

Tipo	Caratteristiche	Template
Polimorfa	Distruttore virtuale	`virtual ~Class() = default;`
Astratta	≥1 metodo virtuale puro	`virtual Type method() = 0;`
Concreta	Implementa tutti i pure virtual	`Type method() override { /* impl */ }`
Base-only	Costruttore protected	`protected: BaseClass();`

⚙ TEMPLATE CLASSE ASTRATTA

```
class AbstractClass {
protected:
    // Costruttore protected se richiesto "non costruibile pubblicamente"
    AbstractClass(/* parametri */);

public:
    // Distruttore virtuale
    virtual ~AbstractClass() = default;

    // Metodi virtuali puri
    virtual ReturnType pureMethod() const = 0;
    virtual ReturnType anotherPureMethod() = 0;

    // Metodi virtuali con implementazione
    virtual ReturnType methodWithImpl() const {
        // Implementazione di default
    }
};
```

⚙ TEMPLATE CLASSE CONCRETA

```
class ConcreteClass : public AbstractClass {
public:
    // Costruttore pubblico
    ConcreteClass(/* parametri */) : AbstractClass(/* parametri */) {}

    // Implementazione metodi virtuali puri (OBBLIGATORIO)
    ReturnType pureMethod() const override {
        // Implementazione concreta
    }

    ReturnType anotherPureMethod() override {
        // Implementazione concreta
    }

    // Override opzionali
    ReturnType methodWithImpl() const override {
        // Implementazione specifica (opzionale)
    }
};
```

REGOLE COSTRUTTORI

Specifica	Implementazione
"Non costruibile pubblicamente"	`protected:` constructor
"Solo come sottooggetto"	`protected:` constructor
"Concreta normale"	`public:` constructor
"Copia e assegnazione"	Implementa secondo templates sezione 5

CHECKLIST GERARCHIE

- ☐ Ogni classe ha tipo corretto (astratta/concreta)
- ☐ Metodi virtuali puri dove richiesti
- ☐ Implementazioni pure virtual in concrete
- ☐ Costruttori con visibilità corretta
- ☐ Distruttore virtuale nella base
- ☐ Definizioni standard se richieste

---

7. STAMPE NUMERICHE

METODOLOGIA STEP-BY-STEP

```
STEP 1: Identifica stringhe certe dalle stampe ("X", "Y")
STEP 2: Mappa stringhe → condizioni nel codice
STEP 3: Deduce gerarchia dalle condizioni
STEP 4: Verifica che tutte le chiamate compilino
STEP 5: Completa il main con chiamate appropriate
```

⚙ PATTERN RICONOSCIMENTO

```
// Pattern tipico funzione F
char F(const A& x, B* y) {
    // Analizza ogni if/else e cosa triggera le stampe
    B* p = const_cast<B*>(dynamic_cast<const B*>(&x));
    auto q = dynamic_cast<const C*>(&x);

    if (dynamic_cast<B*>(y)) {
        if (!p || q) return '1';  // Stampa prima stringa
        else return '2';          // Stampa seconda stringa
    }
    // ... altri controlli
}
```

ALGORITMO DEDUZIONE

1. **Mappa output noto** (es. stampa 6544233241)
2. **Identifica pattern** nelle cifre
3. **Deduce condizioni** che generano ogni cifra
4. **Costruisci gerarchia** compatibile
5. **Scrivi main** che produce l'output

⚙ TEMPLATE MAIN

```
int main() {
    // Crea oggetti necessari basati sulla gerarchia dedotta
    ConcreteType1 obj1;
    ConcreteType2 obj2;
    // ...

    // Chiamate che producono output richiesto
    cout << F(obj1, &obj2) << F(obj2, &obj1) << /* ... */;

    return 0;
}
```

☐ CHECKLIST STAMPE NUMERICHE

- ☐ Pattern output analizzato
- ☐ Condizioni funzione F mappate
- ☐ Gerarchia dedotta correttamente
- ☐ Main produce output esatto
- ☐ Tutte le chiamate compilano

☐ ERRORI COMUNI DA EVITARE

☐ ERRORI GENERALI

- **Dimenticare `virtual` destructor** → memoria non liberata
- **Confondere override con hiding** → chiamate sbagliate
- **Ignorare const-correctness** → non compila
- **Memory leak** nei container → implementa distruttore

☐ ERRORI COSA STAMPA

- **Non verificare signature esatte** → override fallito
- **Ignorare effetto return this** → tipo sbagliato
- **Non tracciare cast** → analisi errata

☐ ERRORI FUNZIONI

- **Iteratori sbagliati** → `const_iterator` vs `iterator`
- **Controlli tipo sbagliati** → `typeid` vs `dynamic_cast`
- **Firma funzione sbagliata** → const missing
- **Memory leak con erase** → delete prima di erase
- **Iteratore invalidato** → aggiornare dopo erase

☐ ERRORI TEMPLATE

- **typename vs class** → preferire `typename`
- **Specializzazione incompleta** → errori di linking
- **Template non istanziato** → no errori compile-time visibili
- **Nome dipendente** → usare `typename` per nested types

☐ ERRORI LAMBDA

- **Capture sbagliato** → [=] vs [&] vs [var]
- **Capture variabili temporanee** → dangling reference
- **Return type mancante** → deduzione sbagliata
- **Mutable dimenticato** → modifica capture by value

☐ ERRORI ALGORITMI STL

- **Dimenticare erase dopo remove_if** → container corrotto
- **Non ordinare prima di unique** → duplicati non rimossi
- **Iteratori invalidati** → crash dopo modifica container
- **Header mancante** → `#include <algorithm>`

⚡ STRATEGIE VELOCI

♂ APPROCCIO RAPIDO COSA STAMPA

1. **Disegna gerarchia** (2 min)
2. **Traccia chiamate principali** (3 min)
3. **Identifica pattern** (2 min)
4. **Verifica edge cases** (3 min)

♂ APPROCCIO RAPIDO IMPLEMENTAZIONE

1. **Template copy-paste** (1 min)
2. **Adatta parametri** (2 min)
3. **Implementa logica core** (5 min)
4. **Test mental** (2 min)

♂ PRIORITÀ TEMPO AGGIORNATA

1. **Cosa stampa semplici** → massimo punteggio/tempo
2. **Sottotipi** → veloci se sai il pattern
3. **Funzioni STL** → template + algoritmi standard
4. **Template semplici** → pattern copy-paste
5. **Lambda con algoritmi** → sintassi standard
6. **Modellazioni** → richiedono più tempo
7. **Gerarchie complesse** → ultime se hai tempo

♂ APPROCCIO RAPIDO TEMPLATE/LAMBDA

1. **Identifica pattern** (funzione vs classe template) (30s)
2. **Copy-paste template base** (1 min)
3. **Adatta parametri** (1 min)
4. **Testa istanziazione** (30s)

♂ APPROCCIO RAPIDO ALGORITMI STL

1. **Identifica operazione** (find/count/transform/remove) (30s)
2. **Scegli algoritmo** dalla tabella (30s)
3. **Implementa lambda/predicato** (2 min)
4. **Verifica gestione iteratori** (1 min)

MATERIALE DI RIFERIMENTO RAPIDO

SYNTAX ESSENZIALE AGGIORNATA

```
// Cast
static_cast<Type*>(ptr)     // Conversione forzata
dynamic_cast<Type*>(ptr)    // Conversione sicura
const_cast<Type*>(ptr)      // Rimozione const

// Type checking
typeid(*ptr) == typeid(Type)              // Tipo esatto
dynamic_cast<Type*>(ptr) != nullptr       // Compatibilità

// Iteratori
vector<T*>::iterator                      // Modifica
vector<T*>::const_iterator               // Sola lettura
auto it = container.begin()              // Auto-deduzione

// Template
template <typename T>                     // Type parameter
template <int N>                         // Non-type parameter
template <>                              // Specializzazione completa

// Lambda
[](params) { return value; }             // Lambda base
[=](params) { /* capture by value */ }   // Capture all by value
[&](params) { /* capture by ref */ }     // Capture all by reference
[var](params) { /* capture var */ }      // Capture specific variable

// Algoritmi STL
find_if(begin, end, predicate)           // Trova con condizione
count_if(begin, end, predicate)          // Conta con condizione
remove_if(begin, end, predicate)         // Marca per rimozione
container.erase(remove_if(...), end)     // Erase-remove idiom
```

🎯 FORMULA DEL SUCCESSO AGGIORNATA

PRATICA + TEMPLATE + SISTEMATICITÀ + ALGORITMI STL = VOTO ALTO

1. **Memorizza i template** di tutte le 10 categorie
2. **Pratica algoritmi STL** con lambda comuni
3. **Automatizza pattern template** più frequenti
4. **Cronometra le soluzioni** per ottimizzare tempo
5. **Rivedi errori comuni** prima dell'esame

RICORDA:

- **Esame moderno** include sempre template/lambda/STL
- **Algoritmi STL** spesso più veloci di implementazioni manuali
- **Lambda** preferite a funtori per logica semplice
- **Template** seguono pattern fissi, non improvvisare