

Esercizio 2.5.1. Quali delle istruzioni numerate nel seguente main() sono legali (cioè compilano) e quali sono illegali?

```
class C {
private:
    int x;
public:
    C(int n = 0) {x=n;}
    C F(C obj) {C r; r.x = obj.x + x; return r;}
    C G(C obj) const {C r; r.x = obj.x + x; return r;}
    C H(C& obj) {obj.x += x; return obj;}
    C I(const C& obj) {C r; r.x = obj.x + x; return r;}
    C J(const C& obj) const {C r; r.x = obj.x + x; return r;}
};

int main() {
    C x, y(1), z(2); const C v(2);
    z=x.F(y);           // (1)
    v.F(y);            // (2)
    v.G(y);            // (3)
    (v.G(y)).F(x);    // (4)
    (v.G(y)).G(x);    // (5)
    x.H(v);            // (6)
    x.H(z.G(y));      // (7)
    x.I(z.G(y));      // (8)
    x.J(z.G(y));      // (9)
    v.J(z.G(y));      // (10)
}
```

```
int main() {
    C x, y(1), z(2); const C v(2);
    z=x.F(y);           // (1): OK
//v.F(y);            // (2): Illegale
    v.G(y);            // (3): OK
    (v.G(y)).F(x);    // (4): OK
    (v.G(y)).G(x);    // (5): OK
//x.H(v);            // (6): Illegale
//x.H(z.G(y));      // (7): Illegale (nota bene!)
    x.I(z.G(y));      // (8): OK (nota bene!)
    x.J(z.G(y));      // (9): OK
    v.J(z.G(y));      // (10): OK
}
```

Esercizio 2.7.3. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

```
class Puntatore {
public:
    int* punt;
};

int main() {
    Puntatore x, y;
    x.punt = new int(8);
    y = x;
    cout << "*("y.punt) = " << *y.punt) << endl;
    *y.punt = 3;
    cout << "*("x.punt) = " << *(x.punt) << endl;
}
```

Stampe:

*(y.punt) = 8
*(x.punt) = 3

Esercizio 2.7.4. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

```
class C {
public:
    C() {}
    C(const C& r) {cout << "*";}
};

C f(C a) {
    C b(a); C c = b; return c;
}

int main() {
    C x;
    C y = f(f(x));
}
```

Stampe:

*

*

*

*

*

Esercizio 2.7.5. Definire una classe Point i cui oggetti rappresentano un punto nello spazio tridimensionale reale. Includere un costruttore di default, un costruttore a tre argomenti

che inizializza un punto, un metodo `negate()` che trasforma un punto nel suo opposto, un metodo `norm()` che restituisce la distanza di un punto dall'origine e un metodo `print()` che stampa le coordinate di un punto. Separare interfaccia ed implementazione della classe.

```
#include <math.h>
#include <iostream>
using std::cout;

class Point {
private:
    double x_, y_, z_;
public:
    Point();
    Point(double, double, double);
    void negate();
    double norm() const;
    void print() const;
};

Point::Point() {x_=0; y_=0; z_=0;}

Point::Point(double x, double y, double z) {x_=x; y_=y; z_=z;}

void Point::negate() {x_ *= -1; y_ *= -1; z_ *= -1;}

double Point::norm() const {return sqrt(x_*x_ + y_*y_ + z_*z_);}

void Point::print() const {cout << '(' <<x_<<', '<<y_<<', '<<z_<<')';}
```

Esercizio 2.7.6. Definire una classe Persona i cui oggetti rappresentano anagraficamente un personaggio storico caratterizzato da nome, anno di nascita e anno di morte. Includere opportuni costruttori, metodi di accesso ai campi e l'overloading dell'operatore di output come funzione esterna. Separare interfaccia ed implementazione della classe. Si definisca inoltre un esempio di metodo `main()` che usa tutti i metodi della classe e l'operatore di output.

```
#include <iostream>

#include "Persona.h"

Persona::Persona(const std::string& name, int n, int m): nome(name), anno_nascita(n), anno_morte(m) {}

std::string Persona::getNome() const{
    return nome;
}

int Persona::getNascita() const{
    return anno_nascita;
```

```

}

int Persona::getMorte() const{
    return anno_morte;
}

std::ostream& operator<<(std::ostream& os, const Persona &p){
    return os << "Nome" << p.nome << "Data di nascita: " << p.anno_nascita << "Data di morte:
" << p.anno_morte;
}

int main(){
    std::string tommy("tommy");
    Persona p1("Mario", 1980, 2010);
    return 0;
}

#endif PERSONA_H
#define PERSONA_H
#include <iostream>

class Persona{
public:
    explicit Persona(const std::string& nome="", int n=0, int m=0);
    int getNascita() const;
    int getMorte() const;
    std::string getNome() const;
    friend std::ostream& operator <<(std::ostream& os, const Persona&);

private:
    std::string nome;
    int anno_nascita, anno_morte;
};

#endif

```

Esercizio 2.8.1. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

```
class C {
private:
    int x;
public:
    C() {cout << "C0 "; x=0;}
    C(int k) {cout << "C1 "; x=k;}
};

class D {
private:
    C c;
public:
    D() {cout << "D0 "; c = C(3);}
};

class E {
private:
    char c;
    C c1;
public:
    D d;
    C c2;
};

int main() {
    D y; cout << endl;           // Stampa: C0 D0 C1
    E x; cout << endl;           // Stampa: C0 C0 D0 C1 C0
    E* p = &x; cout << endl; // Def. di puntatore => nessuna stampa
    D& a = y; cout << endl;   // Def. di riferim. => nessuna stampa
}
```

Esempio 2.8.2. Il seguente programma compila?

```
class D {
private:
    int x;
public:
    D(int a) {x=a;}
};

class C {
private:
    int x;
    D d;
public:
    C() {x=5; d=D(4);}
};
```

```

int main() {
    C z; // Errore in compilazione:
          // In method C::C() no matching function for call to D::D()
}

```

Esercizio 2.10.2. Definire un costruttore di default legale per la classe C.

```

class E {
private:
    int x;
public:
    E(int z=0): x(z) {}
};

class C {
private:
    int z;
    E x;
    const E e;
    E& r;
    int* const p;
public:
    C();
};

```

Soluzione (Possibile) Esercizio 2.10.2.

```
C::C(): z(0), e(E()), r(x), p(new int(0)) {}
```

Esercizio 2.10.3. Il seguente programma compila correttamente (si intende la compilazione g++ con l'opzione -fno-elide-constructors). Quali stampe provoca la sua esecuzione?

```

class D {
private:
    int z;
public:
    D(int x=0): z(x) {cout << "D01 ";}
    D(const D& a): z(a.z) {cout << "Dc ";}
};

class C {
private:
    D d;
public:
    C(): d(D(5)) {cout << "C0 ";}
    C(int a): d(5) {cout << "C1 ";}
    C(const C& c): d(c.d) {cout << "Cc ";}
};

```

```

};

int main() {
    C c1; cout << "UNO" << endl;
    C c2(3); cout << "DUE" << endl;
    C c3(c2); cout << "TRE" << endl;
}

```

Stampa:

D01 Dc C0 UNO

D01 C1 DUE

Dc Cc TRE

Esercizio 2.10.4. Si osservino e confrontino le stampe provocate dall'esecuzione dei seguenti due programmi.

```

class C {
public:
    C() {cout << "C0 ";}
    C(const C&) {cout << "Cc ";}
};

class D {
public:
    C c;
    D() {cout << "D0 ";}
};

int main() {
    D x; cout << endl;
    // stampa: C0 D0
    D y(x); cout << endl;
    // stampa: Cc
}

```

```

class C {
public:
    C() {cout << "C0 ";}
    C(const C&) {cout << "Cc ";}
};

class D {
public:
    C c;
    D() {cout << "D0 ";}
    D(const D&) {cout << "Dc ";}
};

int main() {
    D x; cout << endl;
    // stampa: C0 D0
    D y(x); cout << endl;
    // stampa: C0 Dc
}

```

Si noti attentamente che la differenza è spiegata dal fatto che il costruttore di copia standard di una classe D invoca ordinatamente per ogni campo dati x di D il corrispondente costruttore di copia del tipo di x.

Esercizio 3.6.1. I seguenti programmi compilano correttamente e sono compilati con l'opzione -fno-elide-constructors. Quali stampe produce la loro esecuzione? Attenzione: senza l'opzione -fno-elide-constructors potrebbero produrre un output diverso.

```
// PROGRAMMA UNO

class C {
public:
    string s;
    C(string x="1") : s(x) {}
    ~C() {cout << s << "Cd ";}
};

// funzione esterna, passaggio del parametro per valore
C F(C p) { return p; }

C w("3"); // variabile globale

class D {
public:
    static C c; // campo dati statico
};
C D::c("4");

int main() {
    cout << "PROGRAMMA UNO\n";
    C x("5"), y("6"); D d;
    y=F(x); cout << "uno\n";
    C z=F(x); cout << "due\n";
}
```

// 2 distruttori invocati. uno per il parametro passato e uno per il parametro di ritorno

/*

Programma 1

5Cd 5Cd uno

5Cd 5Cd due

5Cd 5Cd 5Cd 4Cd 3Cd

*/

```
// PROGRAMMA DUE

class C {
public:
    string s;
    C(string x="1"): s(x) {}
    ~C() {cout << s << "Cd ";}
};

// passaggio del parametro per riferimento
C F(C& p) { return p; }

C w("3");

class D {
public:
    static C c;

};

C D::c("4");

int main() {
    cout << "PROGRAMMA DUE\n";
    C x("5"), y("6"); D d;
    y=F(x); cout << "uno\n";
    C z=F(x); cout << "due\n";
}
```

/*

Programma 2

5Cd uno

due

5Cd 5Cd 5Cd 4Cd 3Cd

*/

```

// PROGRAMMA TRE

class C {
public:
    string s;
    C(string x="1") : s(x) {}
    ~C() {cout << s << "Cd ";}
};

// passaggio del parametro e valore ritornato per riferimento
C& F(C& p) { return p; }

C w("3");

class D {
public:
    static C c;
};

C D::c("4");

int main() {
    cout << "PROGRAMMA TRE\n";
    C x("5"), y("6"); D d;
    y=F(x); cout << "uno\n";
    C z=F(x); cout << "due\n";
}

```

/*

Programma 3

uno

due

5Cd 5Cd 5Cd 4Cd 3Cd

*/

```

// PROGRAMMA QUATTRO

class A {
private:
    int z;
public:
    ~A() {cout << "Ad ";}
};

```

```
class B {
public:
    A* p;
    A a;
    ~B() {cout << "Bd ";}
};

class C {
public:
    static B s;
    int k;
    A a;
    ~C() {cout << "Cd ";}
};

B C::s=B();

int main() {
    C c1, c2;
}
```

/*

Bd Ad UNO

DUE

TRE

Cd Ad Cd Ad Cd Ad Bd Ad

*/

Esempio 3.8.1.

```
class C {
public:
    int i;
    C(int x=3): i(x) {cout << "C01 ";}
};

int main() {
    C a[4]; cout << endl;           // stampa: C01 C01 C01 C01
    C* ad = new C[2]; cout << endl; // stampa: C01 C01
    cout << a[0].i << a[1].i << a[2].i << a[3].i << endl;
                                         // stampa: 3333
    cout << ad->i << (ad+1)->i << endl; // stampa: 33
}
```

Per un array a di tipo C è possibile costruire mediante il costruttore di copia gli oggetti di a a partire da una lista di inizializzazione di oggetti di C, analogamente a quanto accade per gli array di tipo non classe. Se la lista di inizializzazione non è completa allora per gli elementi dell'array a per cui non viene fornito un oggetto di C per la costruzione di copia, viene richiamato il costruttore di default (che quindi deve essere disponibile). Si consideri il seguente esempio.

```
class C {
public:
    int i;
    C(int x=3): i(x) {}
};

int main() {
    C x; C y(5);
    C a[4] = {x, C(), y};
    cout << a[0].i << a[1].i << a[2].i << a[3].i << endl;
    // stampa: 3353
}
```

```
// Esempio di narrowing conversion
double d = 3.14;
int x = static_cast<int>(d);
// Esempio di castless conversion
char c = 'a';
int x = static_cast<int>(c);
// Esempio di conversione void* => T*
void* p; p=&d;
double* q = static_cast<double*>(p);
// Esempio di castless conversion
int* r = static_cast<int*>(q);
```

Esercizio 3.11.1. Ridefinire l'operatore “telefonata& operator*()” come metodo della classe iteratore cosicché la funzione Somma_Durate possa essere scritta nel seguente modo:

```
orario Somma_Durate(const bolletta& b) {  
    orario durata;  
    for (bolletta::iteratore it = b.begin(); it != b.end(); it++)  
        durata = durata + (*it).Fine() - (*it).Inizio();  
    return durata;  
}
```

```
telefonata& operator*(telefonata& t, int n) //OPERATORE *  
{  
    orario aux;  
    aux = t.Inizio() + orario(0, 0, n);  
    t.Fine = aux;  
    return t;  
}
```

Esercizio 3.13.1. Aggiungere a bolletta un metodo pubblico

```
void Sostituisci(const telefonata& t1, const telefonata& t2);
```

che modifica la bolletta di invocazione sostituendo la prima (eventuale) occorrenza della telefonata t_1 con la telefonata t_2 . Tale definizione deve gestire la memoria in modo controllato. (*Suggerimento:* fare la copia della bolletta sino a t_1)

Un esercizio più difficile consiste nel definire un metodo

```
void Sostituisci_Tutte(const telefonata& t1, const telefonata& t2);
```

che modifica la bolletta di invocazione sostituendo tutte le (eventuali) occorrenze della telefonata t_1 con la telefonata t_2 . Naturalmente, la memoria deve essere gestita in modo controllato.

```
void bolletta::Sostituisci(const telefonata& t1, const telefonata& t2)
```

```
{  
    smartp p=first, prec, q;  
    smartp original=first;  
    first=0;  
    while(p!=0 && !(p->info==t1))  
    {  
        q=new nodo(p->info, p->next);  
        if (original==p) first=q;  
        else prec->next=q;  
        prec=q;  
        p=q->next;  
    }  
    if (original==p) first=0;  
    else prec->next=0;  
    p=first;  
    while(p!=0)  
    {  
        p->next=original->next;  
        original->next=p;  
        original=p;  
        p=p->next;  
    }  
}
```

```

if(prec==0) first=q;
else prec->next=q;
prec=q; p=p->next;
}

if(p==0) {first=original;}
else if(prec==0) first->info=t2;
else p->info=t2;

}

void bolletta::Sostituisci_Tutte(const telefonata& t1, const telefonata& t2) {
    smartp p = first, prec, q;
    smartp original = first;
    first = 0;
    bool found = false;
    while (p != 0) {
        if (p->info == t1) {
            q = new nodo(t2, p->next);
            if (prec == 0) first = q;
            else prec->next = q;
            prec = q;
            found = true;
            p = p->next;
        } else {
            q = new nodo(p->info, p->next);
            if (prec == 0) first = q;
            else prec->next = q;
            prec = q;
            p = p->next;
        }
    }
    if (!found) { // if t1 is not found, restore original list
        first = original;
    }
}

```

```
 }  
 }
```

Esempio 4.1.1. Ricordiamo che un parametro di una funzione può anche essere un *riferimento ad un array statico*. In questo caso, la dimensione costante dell'array è parte integrante del tipo del parametro e il compilatore controlla che la dimensione dell'array passato come parametro attuale coincida con quella specificata nel tipo del parametro. Consideriamo ad esempio la seguente funzione.

```
int min(int (&a)[7]) { // array di 7 int  
    int m = a[0];  
    for (int i = 1; i < 7 ; i++)  
        if (a[i] < m) m = a[i];  
    return m;  
}
```

È immediato generalizzare questa funzione ad un template con un parametro di tipo ed un parametro valore.

```
template <class T, int size>  
T min(T (&a)[size]) {  
    T vmin = a[0];  
    for (int i = 1; i < size ; i++)  
        if (a[i] < vmin) vmin = a[i];  
    return vmin;  
}  
  
int main() {  
    int ia[20]; orario oa[50];  
    ...  
    cout << min(ia);  
    cout << min(oa);  
    // oppure  
    cout << min<int,20>(ia);  
    cout << min<orario,50>(oa);  
}
```

Esempio 4.3.1. Il seguente programma compila e la sua esecuzione provoca le stampe riportate.

```
// dichiarazione incompleta del template di classe C
template<class T> class C;

// dichiarazione del template di funzione f_friend
template<class T>
void f_friend(C<T>);

template<class T>
class C {
friend void f_friend<T>(C<T>);
private:
    T t;
public:
    C(T x) : t(x) {}
};

template<class T>
void f_friend(C<T> c){
    cout << c.t << endl; // per amicizia
}

int main() {
    C<int> c1(1); C<double> c2(2.5);
    f_friend(c1); // stampa: 1
    f_friend(c2); // stampa: 2.5
}
```

Esempio 4.3.2. Il seguente programma compila e la sua esecuzione provoca le stampe riportate.

```
template <class T>
class C {
    template <class V>
    friend void fun(C<V>);
private:
    T x;
public:
    C(T y) : x(y) {}
};

template <class T>
void fun(C<T> t) {
    cout << t.x << " ";
    C<double> c(3.1);
    cout << c.x << endl; // ok grazie all'amicizia non associata
}

int main() {
    C<int> c(4);
    C<string> s("blob");
    fun(c); // stampa: 4 3.1, istanziazione implicita fun<int>
    fun(s); // stampa: blob 3.1, istanziazione implicita fun<string>
}
```

Esempio 4.3.3. Il seguente programma compila e la sua esecuzione provoca le stampe riportate.

```
template<int I> // parametro valore
class C {
static int numero;
public:
    C();
    void stampa_numero();
};

// inizializzazione parametrica del campo dati statico
template<int I>
int C<I>::numero = I;

template<int I>
C<I>::C() { numero++; }

template<int I>
void C<I>::stampa_numero()
{ cout << "Valore statico: " << numero << endl; }

int main() {
    C<1> uno; C<2> due_a, due_b;
    uno.stampa_numero();    // stampa: 2
    due_a.stampa_numero(); // stampa: 4
    due_b.stampa_numero(); // stampa: 4
}
```

Esercizio 4.3.4. Il seguente programma compila. Quali stampe provoca la sua esecuzione?

```
class A {
public:
    A(int x=0) { cout << x << "A() "; }
};

template<class T>
class C {
public:
    static A s;
};

template<class T>
A C<T>::s=A();

int main() {
    C<double> c;
    C<int> d;
    C<int>::s = A(2);
}
```

Stampe: 0 A() 2 A()

Esercizio 4.4.1. Definire il metodo di cancellazione:

```
template<class T> void Delete(T);
```

che rimuove un nodo contenente un dato valore, se un tale nodo esiste. Si tratta della funzione più complicata, studiata dettagliatamente in un corso di Algoritmi e Strutture Dati.

```
#include <iostream>

using namespace std;

template <class T>
struct Nodo{
    T info;
    Nodo<T>* sinistro, * destro, * padre;
    Nodo(T i, Nodo<T>* s=0, Nodo<T>* d=0, Nodo<T>* p=0):info(i), sinistro(s),
destro(d), padre(p){}
};

template <class T>
class AlbBinRic{
public:
    AlbBinRic():radice(0){}
    ~AlbBinRic(){cancellaR(radice);}
    Nodo<T>* Find(T x) const;
    Nodo<T>* Minimo() const;
    Nodo<T>* Massimo() const;
    Nodo<T>* Succ(Nodo<T>* x) const;
    Nodo<T>* Pred(Nodo<T>* x) const;
    void Insert(T);
    void Delete(T);
    static T Valore(Nodo<T>* p) {return p->info;}
private:
    Nodo<T>* radice;
    void cancellaR(Nodo<T>* p){
        if(p){
            cancellaR(p->sinistro);
            cancellaR(p->destro);
            delete p;
        }
    }
    static Nodo<T>* FindRic(T, Nodo<T>* );
    static Nodo<T>* MinimoRic(Nodo<T>* );
    static Nodo<T>* MassimoRic(Nodo<T>* );
    static void InsertRic(Nodo<T>*, T);
};

//dichiarazione di operator <<
template <class T> ostream& operator<<(ostream&, AlbBinRic<T>');
```

```

#include <iostream>
#include "AlbBinRic.h"

using namespace std;

template <class T>
ostream& operator<<(ostream& os, Nodo<T>& p){
    if(!p) os << "@"; //caso base
    else{
        os << p.info << " ";
        os << p.sinistro << " ";
        os << p.destro;
    }
    return os;
}

template <class T>
Nodo<T>* AlbBinRic<T>::Find(T x) const{
    return FindRic(x, radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::FindRic(T x, Nodo<T>* p){
    if(!p) return 0;
    if(x==p->info) return p;
    if(x < p->info) return FindRic(x, p->sinistro);
    return FindRic(x, p->destro);
}

template <class T>
Nodo<T>* AlbBinRic<T>::Massimo() const{
    if(!radice) return 0;
    return MassimoRic(radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::MassimoRic(Nodo<T>* p){
    if(!p->destro) return p;
    return MassimoRic(p->destro);
}

template <class T>
Nodo<T>* AlbBinRic<T>::Minimo() const{
    if(!radice) return 0;
    return MinimoRic(radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::MinimoRic(Nodo<T>* p){
    if(!p->sinistro) return p;

```

```

        return MinimoRic(p->sinistro);
    }

template <class T>
Nodo<T>* AlbBinRic<T>::Succ(Nodo<T>* x) const{
    if(!x) return 0;
    if(x->destro) return MinimoRic(x->destro);
    //caso x->destro==0
    while(x->padre && x->padre->destro==x) x=x->padre;
    return x->padre;
}

template <class T>
Nodo<T>* AlbBinRic<T>::Pred(Nodo<T>* x) const{
    if(!x) return 0;
    if(x->sinistro) return MassimoRic(x->sinistro);
    //caso x->sinistro==0
    while(x->padre && x->padre->sinistro==x) x=x->padre;
    return x->padre;
}

template <class T>
void AlbBinRic<T>::Insert(T x){
    if(!radice) radice=new Nodo<T>(x);
    else InsertRic(radice, x);
}

template <class T>
void AlbBinRic<T>::InsertRic(Nodo<T>* p, T x){
    if(x < p->info){
        if(!p->sinistro) p->sinistro=new Nodo<T>(p, x);
        InsertRic(p->sinistro, x);
    }
    else{
        if(!p->destro) p->destro=new Nodo<T>(p, x);
        InsertRic(p->destro, x);
    }
}

template <class T>
ostream& operator<<(ostream& os, const AlbBinRic<T>& a) {
    os << *(a.radice);
    return os;
}

template <class T>
void AlbBinRic<T>::Delete(T x){
    Nodo<T>* p=Find(x);
    if(!p) return;
}

```

```

if(!p->sinistro && !p->destro){
    if(!p->padre) radice=0;
    else if(p->padre->sinistro==p) p->padre->sinistro=0;
    else p->padre->destro=0;
    delete p;
}
else if(!p->sinistro || !p->destro){
    Nodo<T>* figlio=p->sinistro ? p->sinistro : p->destro;
    if(!p->padre) radice=figlio;
    else if(p->padre->sinistro==p) p->padre->sinistro=figlio;
    else p->padre->destro=figlio;
    figlio->padre=p->padre;
    delete p;
}
else{
    Nodo<T>* s=MinimoRic(p->destro);
    p->info=s->info;
    if(s->padre->sinistro==s) s->padre->sinistro=s->destro;
    else s->padre->destro=s->destro;
    if(s->destro) s->destro->padre=s->padre;
    delete s;
}
}

int main(){
    return 0;
}

```

Esercizio 5.5.1. Definire un template di funzione che estrae (e quindi anche rimuove) il minimo elemento da un insieme memorizzato in un vector usando solamente il metodo `pop_back()`. Si assuma che sul tipo di base del vector sia definito l'operatore di confronto `operator<` e l'ordine indotto sia totale.

Soluzione (Possibile) Esercizio 5.5.1.

```

template <class T>
T Extract_Min(vector<T> &v) {
    //NOTA: passaggio per riferimento non costante
    if (v.empty()) {cerr << "Coda vuota!"; exit(1);}
    for (vector<T>::iterator i = v.begin()+1; i != v.end(); i++)
        if (*i < *(i-1)) swap<T> (i,i-1);
    T m = *(v.end()-1);
    v.pop_back();
    return m;
}

```

```
template <class T>
void swap(vector<T>::iterator x,vector<T>::iterator y) {
    T t = *x; *x = *y; *y = t;
}
```

```
class C {
private:
    int i;
protected:
    char c;
public:
    float f;
};

class D: private C { }; // derivazione privata
class E: protected C { }; // derivazione protetta

int main() {
    C c, *pc; D d, *pd; E e, *pe;
    c=d;      // Illegale
    c=e;      // Illegale
    pc=&d;    // Illegale
    pc=&e;    // Illegale
    C& rc=d; // Illegale
}
```

```

class C {
private:
    int priv;
protected:
    int prot;
public:
    int publ;
};

class D: private C {
    // prot e publ divengono qui privati
};

class E: protected C {
    // prot e publ divengono qui protetti
};

class F: public D {
    // prot e publ sono qui inaccessibili
public:
    void fF(int i, int j){
        prot=i; // Illegale
        publ=j; // Illegale
    }
};

class G: public E {
    // prot e publ rimangono qui protetti
    void fG(int i, int j){
        prot=i; // OK
        publ=j; // OK
    }
};

```

```

class C {
private:
    int i;
public:
    C(): i(1) {}
    void print() {cout << ' ' << i;}
};

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
    void print() {
        C::print(); // l'oggetto di invocazione di C::print() è il
                    // sottooggetto di tipo C dell'oggetto di invocazione
        cout << ' ' << z;
    }
};

int main() {
    C c; D d;
    c.print(); cout << endl; // stampa: 1
    d.print(); cout << endl; // stampa: 1 3.14
}

```

```
class B {
protected:

    int i;
    void protected_printB() const {cout << ' ' << i;}
public:
    void printB() const {cout << ' ' << i;}
};

class D: public B {
private:
    double z;
public:
    static void stampa(const B& b,const D& d) {
        cout << ' ' << b.i; // Illegale:
                           // "B::i is protected in this context"
        b.printB();          // OK
        b.protected_printB(); // Illegale: "B::protected_printB() is
                           // protected within this context"
        cout << ' ' << d.i; // OK
        d.printB();          // OK
        d.protected_printB(); // OK
    }
};
```

```
class C {
private:
    int i;
public:
    C(): i(1) {}
    friend void print(C);
};

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
};
void print(C x) {
    cout << x.i << endl; D d;
    cout << d.z; // Illegale: "z is private within this context"
}
```

```
int main() {
    C c; D d;
    print(c); // stampa: 1
    print(d); // OK, stampa: 1
}
```

```
class C {
    friend class Z;
private:
    int i;
public:
    C(): i(1) {}
    friend void print(C);
};

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
};

class Z {
public:
    void m() {C c; D d; cout << c.i; // OK
               // cout << d.z; // Illegale: "z is private within this context"
    }
};

int main() {
    Z z;
    z.m(); // stampa: 1
}
```

```
class C {
private:
    int i;
public:
    C(): i(1) {}
    friend void print(C);
};

void print(C x) { cout << x.i << endl; }
```

```

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
    friend void print(D);
};

void print(D x) { cout << x.z << endl; }

int main() {
    C c; D d;
    print(c); // stampa: 1
    print(d); // stampa: 3.14
}

```

```

class C {
public:
    int x;
    void f() { x=4; }
};

class D: public C {
public:
    int y;
    void g() { x=5; y=6; }
};

class E: public D {
public:

    int z;
    void h() { x=7; y=8; z=9; }
};

/* VI significa Valore (intero) Imprevedibile */
int main() {
    C c; D d; E e;
    c.f(); d.g(); e.h();

    // PERICOLOSO
    D* pd = static_cast<D*> (&c);

    // errore run-time o stampa: 4 VI
    cout << pd->x << " " << pd->y << endl;

    // PERICOLOSO
    E& re = static_cast<E&> (d);

    // errore run-time o stampa: 5 6 VI
    cout << re.x << " " << re.y << " " << re.z << endl;

    C* pc = &d; pd = static_cast<D*> (pc);           // OK
    cout << pd->x << " " << pd->y << endl;        // stampa: 5 6
    D& rd = e; E& s = static_cast<E&> (rd);         // OK
    cout << s.x << " " << s.y << " " << s.z << endl; // stampa: 7 8 9
}

```

```

class B {
protected:
    int x;
public:
    B() : x(2) {}
    void print() { cout << x << endl; }
};

class D: public B {
private:
    double x; // ridefinizione del campo dati x
public:
    D() : x(3.14) {}
    // ridefinizione di print()
    void print() { cout << x << endl; } // è la x di D
    void printAll() { cout << B::x << ' ' << x << endl; }
};

int main () {
    B b; D d;
    b.print(); // stampa: 2
    d.print(); // stampa: 3.14
    d.printAll(); // stampa: 2 3.14
}

```

----- esempio.

```

class B {
public:
    // overloading di m
    void m(int x) {cout << "B::m(int)";}
    void m(int x, int y) {cout << "B::m(int,int)";}
};

class D: public B {
public:
    // dichiarazione di uso di B::m
    using B::m;

    // ridefinizione di m
    void m(int x) {cout << "D::m(int)";}
    void m() {cout << "D::m()";}
};

```

```

int main () {
    D d;
    d.m(3); // Compila e stampa: D::m(int)
    d.m(); // Compila e stampa: D::m()
    d.m(3,5); // Compila e stampa: B::m(int,int)
    d.B::m(4); // Compila e stampa: B::m(int)
}

```

```
class B {
public:
    int f() const { cout << "B::f()\n"; return 1; }

    int f(string) const { cout << "B::f(string)\n"; return 2; }
};

class D : public B {
public:
    // ridefinizione con la stessa segnatura
    int f() const { cout << "D::f()\n"; return 3; }
};

class E : public B {
public:
```

```
// ridefinizione con cambio del tipo di ritorno
void f() const { cout << "E::f()\n"; }

class H : public B {
public:
    // ridefinizione con cambio lista argomenti
    int f(int) const { cout << "H::f()\n"; return 4; }

int main() {
    string s; B b; D d; E e; H h;
    int x = d.f(); // Stampa: D::f()
    d.f(s);        // Illegale
    x = e.f();      // Illegale
    x = h.f();      // Illegale
    x = h.f(1);    // Stampa: H::f()
}
```

```
class C {
public:
    void f(int x) {}
    void f() {}
};

class D: public C {
    int y;
public:
    void f(int x) {f(); y=3+x;} // Illegale:
                                // "no matching function for D::f()"
};
```

```
class C {
public:
    int x;
    void f() {x=1;}
};

class D: public C {
public:
    int y;
    void f() {std::cout << "*"; y=3; f();}
    // Errore logico: è una ricorsione infinita
};
```

```
int main() {
    D d; d.f(); // compila ma provoca un errore run-time:
}           // provoca uno stack overflow
```

```
class C {
public:
    void f() {}
    void f(int x) {}
};

class D: public C {
public:
    int f() {return 1;} // cambia il tipo di ritorno
};

int main() {
    D d; d.f(); // OK, chiama D::f()
//d.f(2);    // Illegale
}
```

```
class C {
public:
    void f() {cout << "C::f\n";}
};

class D: public C {
public:
    void f() {cout << "D::f\n";} // ridefinizione
};

class E: public D {
public:
    void f() {cout << "E::f\n";} // ridefinizione
};

int main() {
    C c; D d; E e;
    C* pc = &c; E* pe = &e;
    c = d;      // OK: conversione D -> C
    c = e;      // OK: conversione E -> C
    d = e;      // OK: conversione E -> D
    d = c;      // Illegale: C non ha una classe base D
    C& rc=d;   // OK: conversione D -> C
    D& rd=e;   // OK: conversione E -> D
    pc->f();   // OK, stampa: C::f
```

```

pc = pe; // OK: conversione E* => C*
rd.f(); // OK, stampa: D::f
c.f(); // OK, stampa: C::f
pc->f(); // OK, stampa: C::f
}

```

```

class C {
public:
    int i; double a;
    void f(double) { cout << "C::f(double)\n"; }

class D: public C {
public:
    int* a; // nasconde double C::a
    void f(int*) // nasconde void C::f(double)
};

void D::f(int* p){
    int j; double b; int* q;
    j=i; // OK
    q=a; // OK, a in D è di tipo int*
    b=a; // Illegale: in D a è di tipo int*
    b = C::a; // OK, uso dell'operatore di scoping
    cout << "D::f(int*)\n";
}

int main() {
    int n; double x; int* q;
    D d;
    d.i = n; // OK
    d.a = q; // OK
    d.a = x; // Illegale: d.a ha tipo int*
    d.C::a = x; // OK
    d.f(q); // OK, stampa: D::f(int*)
    d.f(x); // Illegale: d.f() richiede un parametro int*
    d.C::f(x); // OK, stampa: C::f(double)
}

```

Esercizio 6.2.2. I seguenti programmi compilano. Quali stampe provocano le loro esecuzioni?

```

// PROGRAMMA UNO
class C {
public:

```

```

int x;
void f() { x=1; }
};

class D: public C {
public:
    int y;
    void f() { C::f(); y=2; }
};

int main() {
    C c; D d; c.f(); d.f();
    cout << c.x << endl;
    cout << d.x << " " << d.y;
}

```

```

// PROGRAMMA DUE
class C {
public:
    int a;
    void fC() { a=2; }
};

class D: public C {
public:
    double a; // ridefinizione
    void fD() { a=3.14; C::a=4; }
};

class E: public D {
public:
    char a; // ridefinizione
    void fE() { a='*'; C::a=5; D::a=6.28; }
};

int main() {
    C c; D d; E e;
    c.fC(); d.fD(); e.fE();
    D* pd = &d; E& pe = e;
    cout << pd->a << ' ' << pe.a << endl;
    cout << pd->a << ' ' << pd->D::a << ' ' << pd->C::a << endl;
    cout << pe.a << ' ' << pe.D::a << ' ' << pe.C::a << endl;
    cout << e.a << ' ' << e.D::a << ' ' << e.C::a << endl;
}

```

Programma 1:

1

1 2

Programma 2:

3.14 *

3.14 3.14 4

* 6.28 5

* 6.28 5

Esempio 6.3.1.

```
class Z {
public:
    Z() {cout << "Z0 ";}
};

class C {
private:
    int x;
public:
    C(int z=1): x(z) {cout << "C01 ";}
};

};

class D: public C {
private:
    int y;
    Z z;
};

int main() {
    D d; // costruttore standard
}
// Stampa: C01 Z0
```

```
class Z {
public:
    Z() {cout << "Z0 ";}
    Z(double d) {cout << "Z1 ";}
};

class C {
private:
    int x;
    Z w;
public:
    C(): w(6.28), x(8) {cout << x << " C0 ";}
    C(int z): x(z) {cout << x << " C1 ";}
};

class D: public C {
private:
    int y;
    Z z;
public:
    D(): y(0) {cout << "D0 ";}
    D(int a): y(a), z(3.14), C(a) {cout << "D1 ";}
};

int main() {
    D d; cout << endl; // Stampa: Z1 8 C0 Z0 D0
    D e(4);           // Stampa: Z0 4 C1 Z1 D1
}
```

Esercizio 6.3.2. Il seguente programma compila correttamente. Quale stampa provoca la sua esecuzione?

<pre>class C { private: int i; protected: int p; public: C() {cout << "C0 ";} C(int x) {cout << "C1 ";} }; class D: private C { protected: int j; public: D(): C(2) {cout << "D0 ";} D(double x) {cout << "D1 ";} }; class E: public C { private: D d; public: int k; E() : C(6) {cout << "E0 ";} };</pre>	<pre>class F: public D { protected: E e; public: F() : D(3.2) {cout << "F0 ";} F(float x) {cout << "F1 ";} }; class G: public E { private: F f; C c; public: G(): E() {cout << "G0 ";} G(char x) {cout << "G1 ";} }; int main() { G g; }</pre>
--	--

Soluzione Esercizio 6.3.2.

C1 C1 D0 E0 C0 D1 C1 C1 D0 E0 F0 C0 G0
--

Esempio 6.3.3. Il confronto tra le stampe prodotte dall'esecuzione dei seguenti programmi mostra il comportamento del costruttore di copia standard.

<pre>class Z { public: Z() {cout << "Z0 ";} Z(const Z& x) {cout << "Zc ";} }; class C {</pre>
--

```

private:
    Z w;
public:
    C() {cout << "C0 ";}
    C(const C& x): w(x.w) {cout << "Cc ";}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0 ";}
};

int main() {
    D d; cout << "UNO\n";
    D e = d; cout << "DUE"; // costruttore di copia standard
}
// Stampa:
// Z0 C0 Z0 D0 UNO
// Zc Cc Zc DUE

```

```

class Z {
public:
    Z() {cout << "Z0 ";}
    Z(const Z& x) {cout << "Zc ";}
};

class C {
private:
    Z w;
public:
    C() {cout << "C0 ";}
    C(const C& x): w(x.w) {cout << "Cc ";}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0 ";}
    D(const D& x) {cout << "Dc ";}
};

int main() {

    D d; cout << "UNO\n";
    D e = d; cout << "DUE"; // costruttore di copia ridefinito
}
// Stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 Dc DUE

```

Esempio 6.3.4. Il confronto tra le stampe prodotte dall'esecuzione dei seguenti programmi mostra il comportamento dell'assegnazione standard.

```
class Z {
public:
    Z() {cout << "Z0 ";}
    Z(const Z& x) {cout << "Zc ";}
    Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};

class C {
protected:
    Z w;
public:
    C() {cout << "C0 ";}
    C(const C& x): w(x.w) {cout << "Cc ";}
    C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0 ";}
    D(const D& x) {cout << "Dc ";}
};

int main() {
    D d; cout << "UNO\n";
    D e; cout << "DUE\n";
    e=d; cout << "TRE";

}

// Stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 D0 DUE
// Z= C= Z= TRE
```

```
class Z {
public:
    int x;
    Z(): x(0) {cout << "Z0 ";}
    Z(const Z& x) {cout << "Zc ";}
    Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};

class C {
public:
    Z w;
    C() {cout << "C0 ";}
    C(const C& x): w(x.w) {cout << "Cc ";}
    C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};

class D: public C {
public:
    Z z;
    D() {cout << "D0 ";}
    D(const D& x) {cout << "Dc ";}
    D& operator=(const D& x) {z=x.z; cout << "D= "; return *this;}
    // l'assegnazione è definita male: chi ci pensa ad assegnare il
    // campo dati w di C? E se w fosse marcato private?
};

int main() {
    D d; d.w.x = 3; cout << "UNO\n";
    D e; e.w.x = 5; cout << "DUE\n";
    e=d; cout << "TRE\n";
    cout << e.w.x << ' ' << d.w.x << " QUATTRO";
}

// Stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 D0 DUE
// Z= D= TRE
// 5 3 QUATTRO
```

Esempio 6.3.5. Il confronto tra le stampe prodotte dall'esecuzione dei seguenti programmi mostra il comportamento del distruttore standard.

```
class Z {
public:
    Z() {cout << "Z0";}
    ~Z() {cout << "~Z";}
};

class C {
protected:
    Z w;
public:
    C() {cout << "C0";}
    ~C() {cout << "~C";}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0";}
};

int main() {
    D* p = new D; cout << "UNO\n";
    delete p; cout << "DUE"; // distruttore standard
}
// Stampa:
// Z0 C0 Z0 D0 UNO
// ~Z ~C ~Z DUE
```

```
class Z {
public:
    Z() {cout << "Z0";}
    ~Z() {cout << "~Z";}
};

class C {

protected:
    Z w;
public:
    C() {cout << "C0";}
    ~C() {cout << "~C";}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0";}
    ~D() {cout << "~D";}
};

int main() {
    D* p = new D; cout << "UNO\n";
    delete p; cout << "DUE"; // distruttore ridefinito
}
// Stampa:
// Z0 C0 Z0 D0 UNO
// ~D ~Z ~C ~Z DUE
```

... naturalmente, questo vale per un metodo qualsiasi, in particolare per gli operatori.
Consideriamo il seguente esempio di gerarchia:

```
class B {
public:
    virtual int f() {cout << "B::f()\n"; return 1;}
```

6.5 Metodi virtuali

```
virtual void f(string s) {cout << "B::f(string)\n";}
virtual void g() {cout << "B::g()\n";}
};

class D1 : public B {
public:
    // Overriding di un metodo virtuale non sovraccaricato
    void g() {cout << "D1::g()\n";}
};

class D2 : public B {
public:
    // Overriding di un metodo virtuale sovraccaricato
    int f() {cout << "D2::f()\n"; return 2;}
};

class D3 : public B {
public:
    // NON è possibile modificare il tipo di ritorno
    void f() {cout << "D3::f()\n"; // Illegale}
};

class D4 : public B {
public:
    // Lista degli argomenti modificata:
    // è una ridefinizione e non un overriding
    int f(int) {cout << "D4::f()\n"; return 4;}
};
```

Consideriamo quindi il seguente esempio di codice che usa le classi della precedente gerarchia

```
int main() {
    string s="ciao"; D1 d1; D2 d2; D4 d4;
    int x = d1.f();      // Stampa: B::f()
    d1.f(s);           // Stampa: B::f(string)
    x = d2.f();         // Stampa: D2::f()
    d2.f(s);           // Illegale: "no matching function"
    x = d4.f(1);       // Stampa: D4::f()
    x = d4.f();         // Illegale: "no matching function"
    d4.f(s);           // Illegale: "no matching function"
    B& br = d4;        // Cast隐式
    br.f(1);           // Illegale
    br.f();            // OK, stampa: B::f()
    br.f(s);           // OK, stampa: B::f(string)
}
```

```

class X {};
class Y: public X {};
class Z: public X {};

class B {
    X x;
public:
    virtual X* m() { cout << "B::m() "; return &x; }
};

class C: public B {
    Y y;
public:
    virtual X* m() { return &y; }
};

class D: public B {
    Z z;
public:
    virtual Z* m() { return &z; } // OK, overriding legale
};

int main() {
    C c; D d;
    Y* py = c.m(); // Illegale
    X* px = c.m();
    Z* pz = d.m();
}

```

```

class B {
public:
    virtual void m(int x=0) { cout << "B::m ";}
};

class D: public B {
public:
    // è un overriding di B::m
    virtual void m(int x) { cout << "D::m ";}

    // Legale: è un nuovo metodo in D e non un overriding di B::m
    virtual void m() { cout << "D::m() ";}
};

int main() {
    B* p = new D;
    D* q = new D;
    p->m(2); // Stampa D::m e non B::m
    p->m(); // Stampa D::m e non D::m()
    q->m(); // Stampa D::m() e non D::m
}

```

```

class B {
public:
    virtual void m(int x=0) { cout << "B::m ";}
};

class D: public B {

public:
    // è un overriding di B::m
    virtual void m(int x=0) { cout << "D::m ";}

    // Legale: è un nuovo metodo in D e non un overriding di B::m
    virtual void m() { cout << "D::m() ";}
};

int main() {
    B* p = new D;
    D* q = new D;
    p->m(2); // Stampa D::m e non B::m
    p->m(); // Stampa D::m e non D::m()
    q->m(); // Illegale: chiamata ambigua di m() sovraccaricato
}

```

Consideriamo la seguente situazione.

```
class B {
public:
    virtual void m() { cout << "B::m() " ; }
};

class C: public B {
public:
    virtual void m() { cout << "C::m() " ; }
};

class D: public C {
public:
    virtual void m() { cout << "D::m() " ; }
};

int main() {
    C* p = new D();
    p->m();      // Dynamic binding, stampa: D::m()
    p->B::m();   // Static binding, stampa: B::m()
    p->C::m();   // Static binding, stampa: C::m()
}
```

```
class B {
private:
    int* p;
public:
    B(int n, int v) : p(new int[n]) {
        for(int i=0; i<n; i++) p[i]=v;
    }
    virtual ~B() {delete[] p; cout << "~B() " ;} // distr. virtuale
};

class C : public B {
private:
    int* q;
public:
    C(int sizeB, int sizeC, int v): B(sizeB,v), q(new int[sizeC]) {
        for(int i=0; i<sizeC; i++) q[i]=v;
    }
    virtual ~C() {delete[] q; cout << "~C() " ;}
};

int main() {
    C* q = new C(4,2,18);
    B* p=q; // puntatore polimorfo
    delete p; // distruzione virtuale: invoca ~C()
}
// Stampa: ~C() ~B()
// Se ~B() non fosse virtuale verrebbe invocato solamente ~B()
```

Consideriamo il seguente esempio:

```
class B { // classe base astratta
public:
    virtual void f() = 0;
};

class C: public B {}; // sottoclasse astratta

class D: public B { // sottoclasse concreta
public:
    virtual void f() {cout << "D::f() ";}
};

int main() {
    C c;      // Illegale: "cannot declare c of type C ..."
    D d;      // OK, D è concreta
    B* p;     // OK, puntatore a classe astratta
    p = &d;   // Puntatore polimorfo
    p->f();  // Stampa: D::f()
}
```

Esempio 6.7.1. Si consideri la seguente gerarchia di classi dove D è definita tramite derivazione multipla.

```
class A {
public:
    int a;
    A(int x=1): a(x) {}
};

class B: public A {
public:
    B(): A(2) {}
};

class C: public A {
public:
    C(): A(3) {}
};

class D: public B, public C {};
```

Il seguente frammento di codice non compila a causa dell'ambiguità generata dalla derivazione multipla.

```
D d;
A* p = &d;      // Illegale: A è una classe base ambigua per D
cout << p->a; // quale sottooggetto di A si dovrebbe usare?
```

Si consideri ora la seguente gerarchia di classi.

```
class A {
public:
    virtual void print() =0; // virtuale puro
};

class B: public A {
public:
    virtual void print() {cout << "B ";} // implementazione
};

class C: public A {
public:
    virtual void print() {cout << "C ";} // implementazione
};

class D: public B, public C {
    virtual void print() {cout << "D ";} // overriding
};
```

In questo caso il seguente frammento di codice non compila ancora a causa dell'ambiguità dovuta all'ereditarietà multipla a diamante.

```
D d;
A* p = &d;      // Illegale: A è una classe base ambigua per D
p->print(); // la chiamata polimorfa non è legale
```

Esempio 6.7.2. Si consideri la seguente gerarchia di classi a diamante definita senza derivazione virtuale.

```
class A { int a[5]; };           // 20 byte
class B: public A { int y[3]; }; // 12 byte
class C : public A { int z[3]; }; // 12 byte
class D: public B, public C {
    int w[4];                  // 16 byte
};
```

```

cout << "sizeof(A) == " << sizeof(A) << endl; // 20
cout << "sizeof(B) == " << sizeof(B) << endl; // 32=12+20
cout << "sizeof(C) == " << sizeof(C) << endl; // 32=12+20
cout << "sizeof(D) == " << sizeof(D);           // 80=16+32+32

```

Consideriamo ora la stessa gerarchia di classi definita con derivazione virtuale.

```

class A {int a[5];} // 20 byte

class B: virtual public A {
    int y[3];          // 12 byte + 1 puntatore
};

class C : virtual public A {
    int z[3];          // 12 byte + 1 puntatore
};

class D: public B, public C {
    int w[4];          // 16 byte
};

```

In questo caso le stampe prodotte dal precedente codice risultano piuttosto diverse:

```

cout << "sizeof(A) == " << sizeof(A) << endl;           // 20
cout << "sizeof(B) == " << sizeof(B) << endl;           // 36=12+4+20
cout << "sizeof(C) == " << sizeof(C) << endl;           // 36=12+4+20
cout << "sizeof(D) == " << sizeof(D); // 68=16+(12+4)+(12+4)+20

```

Esempio 6.7.3. Consideriamo la gerarchia di classi per il secondo programma dell'Esempio 6.7.1, definita in questo caso tramite derivazione virtuale.

```

class A {
public:
    virtual void print()=0;
};

class B: virtual public A {
public:
    virtual void print() {cout << "B ";}
};

class C: virtual public A {
public:
    virtual void print() {cout << "C ";}
};

```

```
class D: public B, public C {  
    virtual void print() {cout << "D ";}  
};
```

In questo caso è necessario definire l'overriding di `print()` in `D`, altrimenti si otterrebbe il seguente errore in compilazione: "no unique final overrider for `A::print()`". Con queste nuove definizioni il seguente codice compila e produce la stampa indicata.

```
D d;  
A* p = &d; // Compila  
p->print(); // Stampa: D
```

Anche per la derivazione virtuale multipla, possiamo avere derivazione privata, pubblica o protetta. Quindi in una classe derivata può accadere di avere la stessa classe base virtuale indiretta ma con diverse regole d'accesso. In un caso del genere prevale la modalità di derivazione più permissiva, ovvero vale la seguente regola: la derivazione protetta prevale su quella privata, e la derivazione pubblica prevale su quella protetta. Ad esempio:

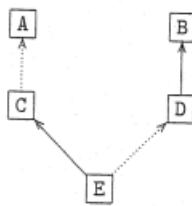
```
class A { public: void f() {cout << "A";} };  
  
class B: virtual private A {}; // derivazione virtuale privata  
  
class C: virtual public A {}; // derivazione virtuale pubblica  
  
class D: public B, public C {};// prevale la derivazione pubblica  
  
int main() {  
    D d;  
    d.f(); // OK, stampa: A (è il metodo C::f())  
//d.B::f(); // Illegale, A::f() inaccessibile  
}
```

```
class A {  
public:  
    int a;  
    A(int x=1): a(x) {}  
};  
  
class B: virtual public A {  
public:  
    B(): A(2) {}  
};  
  
class C: virtual public A {  
public:  
    C(): A(3) {}  
};  
  
class D: public B, public C {};
```

Il seguente codice compila ed esegue correttamente producendo, come ci si aspetta, la stampa riportata.

```
D d;  
A* p = &d; // Compila  
cout << p->a; // Stampa: 1 (e non 2 o 3!)
```

Esempio 6.7.5.



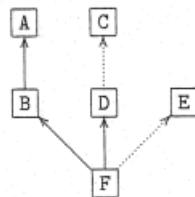
```
class A { public: A() {cout << "A ";} };
class B { public: B() {cout << "B ";} };
class C: virtual public A { public: C() {cout << "C ";} };
class D: public B { public: D() {cout << "D ";} };
```

234

6.7 Ereditarietà multipla

```
class E : public C, virtual public D {
    public: E() {cout << "E ";}
};
int main() { E e; } // stampa: A B D C E
```

Esempio 6.7.6.

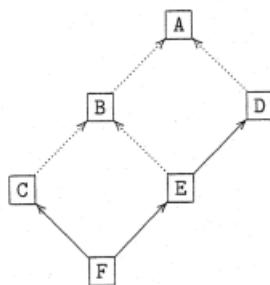


```
class A { public: A() {cout << "A ";} };
class B: public A {
    public: B() {cout << "B ";}
};
class C {
    public: C() {cout << "C ";}
};
class D: virtual public C {
    public: D() {cout << "D ";}
};
class E { public: E() {cout << "E ";} };

class F: public B, public D, virtual public E {
    public: F() {cout << "F ";}
};

int main() { F f; } // stampa: C E A B D F
```

Esempio 6.7.7.

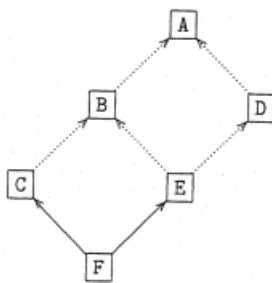


```

class A { public: A() {cout << "A ";}
};
class B: virtual public A {
    public: B() {cout << "B ";}
};
class D: virtual public A {
    public: D() {cout << "D ";}
};
class C: virtual public B {
    public: C() {cout << "C ";}
};
class E: virtual public B, public D {
    public: E() {cout << "E ";}
};
class F: public C, public E {
    public: F() {cout << "F ";}
};
int main() { F f; } // stampa: A B C D E F

```

Esempio 6.7.8.



```

class A { public: A() {cout << "A ";}
};
class B: virtual public A {
    public: B() {cout << "B ";}
};
class D: virtual public A {
    public: D() {cout << "D ";}
};
class C: virtual public B {
    public: C() {cout << "C ";}
};

```

A SOLUZIONI DEGLI ESERCIZI

```
void IntMod::set_modulo(int mod) { modulo=mod; }

IntMod IntMod::operator+(const IntMod& m) const {
    return IntMod(val + m.val);
}

IntMod IntMod::operator*(const IntMod& m) const {
    return IntMod(val * m.val);
}

int IntMod::modulo=1;
```

```
//ESEMPIO DI UTILIZZO
#include <iostream>
#include "intmod.h"
using std::cout; using std::endl;

int main() {
    IntMod::set_modulo(4);
    IntMod a(3), b(2);
    cout << a+b << endl;           // stampa 1
    cout << a*b << endl;           // stampa 2
    cout << IntMod(4)+a << endl; // stampa 3
    cout << a+4 << endl;           // stampa 7
}
```

Soluzione Esercizio 11.2.

```
0 6 7 8
```

Soluzione Esercizio 11.3.

```
0 1 2 3 UNO
0 1 3 6 DUE
```

Soluzione (Possibile) Esercizio 11.4.

```
class Nodo {
friend class Albero;
private:
    Nodo(string st="***", Nodo* s=0, Nodo* d=0): info(st), sx(s),
                                                    dx(d) {}
    string info;
```

A.1 Esercizi del Capitolo 11

```
Nodo* sx;
Nodo* dx;
};

class Albero {
public:
    Albero(): radice(0) {}
    Albero(const Albero& t) {radice = copia(t.radice);}
private:
    Nodo* radice;
    static Nodo* copia(Nodo* p) {
        if (!p) return 0;
        else return new Nodo(p->info, copia(p->sx), copia(p->dx));
    }
};
```

Soluzione (Possibile) Esercizio 11.5.

```
// file "data.h"
#ifndef DATA_H
#define DATA_H
#include <iostream>
#include <string>
using std::string; using std::ostream;

class Data {
public:
    Data(string = "", int = 1, int = 1, int = 0);

    string get_gsett() const;
    int get_giorno() const;
    int get_mese() const;
    int get_anno() const;

    bool operator==(const Data&) const;
    bool operator<(const Data&) const;
    void aggiungi_anno();

private:
    int giorno, mese, anno;
    string gsett;
};

ostream& operator<<(ostream&, const Data&);

#endif
```

```
// file "data.cpp"
```

A SOLUZIONI DEGLI ESERCIZI

```
#include "data.h"

Data::Data(string s, int g, int m, int a):
    gsett(s), giorno(g), mese(m), anno(a) {}

string Data::get_gsett() const {return gsett;}

int Data::get_giorno() const {return giorno;}

int Data::get_mese() const {return mese;}

int Data::get_anno() const {return anno;}

bool Data::operator==(const Data& d) const {
    return ((gsett==d.gsett) && (giorno==d.giorno) &&
            (mese==d.mese) && (anno==d.anno));
}

bool Data::operator<(const Data& d) const {
    if(anno < d.anno) return true;
    if(anno==d.anno && mese<d.mese) return true;
    if ((anno == d.anno) && (mese==d.mese) && (giorno < d.giorno))
        return true;
    return false;
}

ostream& operator<<(ostream& os, const Data& d) {
    return os << d.get_gsett() << " " << d.get_giorno()
        << "/" << d.get_mese() << "/" << d.get_anno();}
```

```
// ESEMPIO DI UTILIZZO
#include "data.h"
#include <iostream>
using std::cout;

int main() {
    Data d("lun",21,10,2002), e("mar",22,10,2002);
    cout << d << endl;
    cout << (d<e) << " " << (e==d) << endl;
}
```

Soluzione (Possibile) Esercizio 11.6.

```
// file "vettore.h"
#ifndef VETTORE_H
```

A.1 Esercizi del Capitolo 11

```
#define VETTORE_H
#include<iostream>
using std::ostream;

class Vettore {
public:
    Vettore(int =1, int =0);
    Vettore(const Vettore& );
    Vettore& operator=(const Vettore& );
    int& operator[](int) const;
    bool operator==(const Vettore&) const;
    int dim() const { return size; } // definizione inline
private:
    int size; // dimensione del vettore
    int* v; // array dinamico che memorizza il vettore
};

ostream& operator<<(ostream&, const Vettore& );
#endif
```

```
// file "vettore.cpp"
#include "vettore.h"
using std::cout; using std::endl;

Vettore::Vettore(int sz, int x): size(sz), v(new int[sz]) {
    for(int i=0; i<size; i++) v[i] = x;
}

Vettore::Vettore(const Vettore& x): size(x.size),
                           v(new int[x.size]) {
    for(int i=0; i<size; i++) v[i]=x.v[i];
}

Vettore& Vettore::operator=(const Vettore& x) {
    if(this != &x) { // se siamo nel caso x=x non faccio nulla
        delete[] v; // dealloco
        size = x.size;
        v = new int[x.size];
        for(int i=0; i<size; i++) v[i]=x.v[i];
    }
    return *this;
}

int& Vettore::operator[](int i) const { return v[i]; }
```

A SOLUZIONI DEGLI ESERCIZI

```
bool Vettore::operator==(const Vettore& x) const {
    if(size != x.size) return false;
    for(int i=0; i<size; i++) if(v[i] != x.v[i]) return false;
    return true;
}

ostream& operator<<(ostream& ostr, const Vettore& x) {
    ostr << '(';
    for(int i=0; i<x.dim()-1; i++) {
        ostr << x[i] << ',';
        if((i+1)%10 == 0) cout << endl;
    }
    return ostr << x[x.dim()-1] << ")\n";
}
```

Soluzione (Possibile) Esercizio 11.7.

```
int main() {
    A a; B b; C c; D d; E e;
    cout << F(&b,e) << F(&b,c) << F(&a,c) << F(&a,e);
}
```

Soluzione (Possibile) Esercizio 11.8.

```
int Fun(vector<B*>& v) {
    int tot = 0;
    for(vector<B*>::iterator it = v.begin(); it!= v.end(); ++it)
        if(typeid(*v[0])!=typeid(*(*it)) && dynamic_cast<C*>(*it))
            ++tot;
    return tot;
}
```

Soluzione (Possibile) Esercizio 11.9.

```
class Auto {
private:
    int cavalliFiscali;
    static double tassaPerCF;
protected:
    Auto(int cf = 0): cavalliFiscali(cf) {}
    virtual ~A() {}
public:
    int getCavalliFiscali() const {return cavalliFiscali;}
    static double getTassaPerCF() {return tassaPerCF;}
    virtual double tassa() const = 0;
```

A.1 Esercizi del Capitolo 11

```
};

double Auto::tassaPerCF = 5;

class Diesel: public Auto {
private:
    static double tassaDiesel;
public:
    Diesel(int cf = 0): Auto(cf) {}
    virtual double tassa() const {
        return getCavalliFiscali() * getTassaPerCF() + tassaDiesel;
    }
};
double Diesel::tassaDiesel = 100;

class Benzina: public Auto {
private:
    static double bonusEco4;
    bool eco4;
public:
    Benzina(int cf = 0, bool x = true): Auto(cf), eco4(x) {}
    virtual double tassa() const {
        return (eco4 ? getCavalliFiscali()*getTassaPerCF()-bonusEco4;
                  : getCavalliFiscali()*getTassaPerCF());
    }
};
double Benzina::bonusEco4 = 50;

class ACI {
private:
    vector<Auto*> v;
public:
    double incassaBolli() const {
        double tot = 0;
        for(vector<Auto*>::const_iterator it=v.begin(); it!=v.end();
            ++it)
            tot += (*it)->tassa();
        return tot;
    }
    void aggiungiAuto(const Auto& a) {
        Auto& t = const_cast<Auto&>(a); v.push_back(&t);
    }
};

int main() {
    Benzina fiat(60, false), lancia(60, true);
```

A SOLUZIONI DEGLI ESERCIZI

```
Diesel bmw(90), audi(80);
ACI a;
a.aggiungiAuto(fiat); a.aggiungiAuto(lancia);
a.aggiungiAuto(bmw); a.aggiungiAuto(audi);
cout << a.incassaBolli() << endl; // stampa: 1600
}
```

Soluzione Esercizio 11.10.

- (1) COMPILA.
- (2) NON COMPILA.
- (3) COMPILA.
- (4) NON COMPILA.
- (5) NON COMPILA.
- (6) COMPILA.

Soluzione (Possibile) Esercizio 11.11.

```
class Biglietto {
private:
    string nome;
    bool galleria;
protected:
    Biglietto(const Biglietto& b) :
        nome(b.nome), galleria(b.galleria) {}
    Biglietto(string n, bool g=false): nome(n), galleria(g) {}
    virtual ~Biglietto() {}
public:
    bool isGalleria() const {return galleria;}
    string getNome() const {return nome;}
};

class PostoNumerato: public Biglietto {
private:
    int fila;
public:
    PostoNumerato(string nome, int f): Biglietto(nome), fila(f) {}
    int numFila() const {return fila;}
};

class PostoNonNumerato: public Biglietto {
private:
```

A.1 Esercizi del Capitolo 11

```
bool ridotto;
public:
    PostoNonNumerato(string nome, bool g=true, bool r=false) :
        Biglietto(nome, g), ridotto(r) {}
    bool Ridotto() const {return ridotto;}
};

class Spettacolo {
private:
    double prezzoBase, addizionale;
    int maxNumerati, maxFila;
    int numeratiVenduti;
    list<Biglietto*> l;
public:
    Spettacolo(double pb, double add, int maxN, int maxF, int nv=0) :
        prezzoBase(pb), addizionale(add), maxNumerati(maxN),
        maxFila(maxF), numeratiVenduti(nv) {}

    void aggiungiBiglietto(const Biglietto& b) {
        Biglietto* p = const_cast<Biglietto*>(&b);
        if(dynamic_cast<PostoNumerato*>(p))
            if(numeratiVenduti < maxNumerati)
                {numeratiVenduti++; v.push_back(p);}
            else cout << "Posti numerati esauriti\n";
        else if(dynamic_cast<PostoNonNumerato*>(p)) l.push_back(p);
    }

    double prezzo(const Biglietto& b) const {
        double s = 0;
        Biglietto* p = const_cast<Biglietto*>(&b);
        PostoNumerato* x = dynamic_cast<PostoNumerato*>(p);
        if(x)
            s = ((x->numFila() <= maxFila) ? 2*prezzoBase+2*addizionale
                                              : 2*prezzoBase);
        else {
            PostoNonNumerato* y = dynamic_cast<PostoNonNumerato*>(p);
            if(y) {
                s = prezzoBase;
                if(!y->isGalleria()) s += addizionale;
                if(y->Ridotto()) s -= addizionale/2;
            }
        }
        return s;
    }
    double incasso() const {
```

A SOLUZIONI DEGLI ESERCIZI

```
double tot=0;
for(list<Biglietto*>::const_iterator it = l.begin();
                                         it!= l.end(); ++it)
    tot += prezzo(**it);
return tot;
};

int main() {
    Spettacolo s(10,6,200,10);
    PostoNumerato a1("pippo",2);           // prezzo: 32
    PostoNumerato a2("pluto",17);         // prezzo: 20
    PostoNonNumerato b1("zagor");        // prezzo: 10
    PostoNonNumerato b2("pluto",false,true); // prezzo: 13
    s.aggiungiBiglietto(a1); s.aggiungiBiglietto(a2);
    s.aggiungiBiglietto(b1); s.aggiungiBiglietto(b2);
    cout << s.incasso();                  // stampa: 75
}
```

Soluzione Esercizio 11.12.

```
8 2 9 UNO
8 2 2 7 DUE
1 3 1 7 TRE
1 3 1 7 QUATTRO
```

Soluzione (Possibile) Esercizio 11.13.

```
class C {
private:
    vector<ios*> v;
    int max;
public:
    C(int k=10): max(k) {}
    void insert(ios& s) {
        if( v.size() < max && typeid(s)!=typeid(stringstream)
            && typeid(s)!=typeid(fstream) )
            v.push_back(&s);
    }
    template<class T>
    int conta(T& t) const {
        int x=0;
        for(vector<ios*>::const_iterator it=v.begin(); it!=v.end();
                                         ++it) {
            if(dynamic_cast<T*>(*it)) ++x;
        }
    }
};
```

A.1 Esercizi del Capitolo 11

```
    }
    return x;
}
};
```

Soluzione Esercizio 11.14.

```
Z() A() Z() B() Z() A() Z() C() **0
Z() A() Z() Bc **1
Z() Ac Zc **2
Z() A() Z() B() Z() C() D() **3
Z() A() Z() B() Zc Dc **4
```

Soluzione (Possibile) Esercizio 11.15.

```
class RitiraPremio {};

class Prodotto {
private:
    double prezzo;
public:
    Prodotto(double p): prezzo(p) {}
    double getPrezzo() const {return prezzo;}
    double setPrezzo(double p) {prezzo=p;}
};

class Cliente {
private:
    vector<Prodotto> carrello;
public:
    virtual ~Cliente() {}
    virtual double spesaTotale() const {
        double tot=0;
        for(int i=0; i<carrello.size(); i++)
            tot += carrello[i].getPrezzo();
        return tot;
    }
};

class ClienteFedele : public Cliente {
private:
    int punti; // saldo punti
    static int sogliaPuntiPremio; // punti richiesti per premio
    static int sconto; // sconto in percentuale
public:
```

A SOLUZIONI DEGLI ESERCIZI

```
ClienteFedele(int p=0): punti(p) {}
virtual double spesaTotale() const {
    double tot=Cliente::spesaTotale();
    tot *= 1-sconto/100;
    return tot;
}
void accreditaPunti(int n) throw(RitiraPremio) {
    punti +=n;
    if(punti>=sogliaPuntiPremio){
        punti -= sogliaPuntiPremio;
        throw RitiraPremio();
    }
}
int saldoPunti() const {return punti;}
};

int ClienteFedele::sogliaPuntiPremio=100;
int ClienteFedele::sogliaPuntiPremio=5;

class GestioneGiornaliera {
private:
    vector<Cliente*> v;
public:
    double chiudiCassa() const {
        double tot=0;
        for(int i=0; i<v.size(); ++i) {
            double s = v[i]->spesaTotale();
            tot += s;
            ClienteFedele* p = dynamic_cast<ClienteFedele*>(v[i]);
            try{p->accreditaPunti(static_cast<int>(s/10));}
            catch(RitiraPremio) {}
        }
        return tot;
    }
    int saldoPuntiGiornaliero() const {
        int tot=0;
        for(int i=0; i<v.size(); ++i) {
            ClienteFedele *p=dynamic_cast<ClienteFedele*>(v[i]);
            if(p) tot+= p->saldoPunti();
        }
        return tot;
    }
};
```

Soluzione (Possibile) Esercizio 11.16.

```
// dichiarazione incompleta
```

A.1 Esercizi del Capitolo 11

```
template <class T, int size> class C;

// dichiarazione incompleta
template <class T, int size>
ostream& operator<< (ostream&, const C<T,size>&);

template <class T=string, int size=1>
class C {
    friend ostream& operator<< <T,size>(ostream&, const C<T,size>&);
public:
    C(const T& t=T(), int k=1): array(new MultiInfo[size](t,k)) {}
    C(const C& x): array(new MultiInfo[size]) {
        for(int i=0; i<size; ++i) array[i] = x.array[i];
    }
    ~C() {delete[] array;}
    C& operator=(const C& x) {
        if (this != &x) {
            // delete[] array; // non serve!
            for(int i=0; i<size; ++i) array[i] = x.array[i];
        }
        return *this;
    }
    T* operator[](int k) const {
        return (0<=k && k<size)? &(array[k].info) : 0;
    }
    int occorrenze(const T& t) const {
        int sum=0;
        for(int i=0; i<size; ++i)
            if(array[i].info==t) sum+=array[i].mult;
        return sum;
    }
private:
    class MultiInfo {
public:
    MultiInfo(const T& x = T(), int z = 1) : info(x), mult(z) {
        if (mult<1) mult=1;
    }
    T info;
    int mult;
    };
    MultiInfo* array;
};

template <class T, int size>
ostream& operator<< (ostream& os, const C<T,size>& x) {
```

A SOLUZIONI DEGLI ESERCIZI

```
for(int i=0; i<size; ++i) os << " Valore: " <<  
    x.array[i].info << " Molteplicita': " << x.array[i].mult;  
return os;  
}
```

Soluzione (Possibile) Esercizio 11.17.

```
class NoButton {};  
  
Button** Fun(const Container& c) throw(NoButton) {  
    vector<Component*> v = c.getComponents();  
    int cont=0;  
    for(int i=0;i<v.size();++i)  
        if(dynamic_cast<Button*>(v[i])) ++cont;  
    if(cont==0) throw NoButton();  
    Button** a = new Button*[cont];  
    cont=0;  
    for(int i=0; i<v.size(); ++i) {  
        if(dynamic_cast<Button*>(v[i])) {  
            MenuItem* mi = dynamic_cast<MenuItem*>(v[i]);  
            if(mi&&mi->getContainers().size()>1) mi->setEnabled(false);  
            a[cont]=dynamic_cast<Button*>(v[i]);  
            ++cont;  
        }  
    }  
    return a;  
}
```

Soluzione (Possibile) Esercizio 11.18.

```
class FileAudio {  
public:  
    virtual FileAudio* clone() const =0;  
    virtual bool qualita() const =0;  
    double dimensione() const {return dim;}  
private:  
    double dim; // MB  
};  
  
class Mp3: public FileAudio {  
public:  
    virtual FileAudio* clone() const {  
        return new Mp3(*this);  
    }  
    virtual bool qualita() const {
```

A.1 Esercizi del Capitolo 11

```
    return BitRate >= 192;
}
bool operator==(const Mp3& x) const {
    return dimensione() == x.dimensione() &&
        bitrate() == x.bitrate();
}
int bitrate() const {return BitRate;}
private:
    int BitRate; // Kbits
};

class WAV: public FileAudio {
public:
    virtual FileAudio* clone() const {
        return new WAV(*this);
    }
    virtual bool qualita() const {
        return frequenzaCampionamento >= 96;
    }
    bool LossLess() const {return lossless;}
    double freq() const {return frequenzaCampionamento;}
private:
    bool lossless;
    double frequenzaCampionamento; // kHz
};

class iZod {
private:
    class Brano {
public:
    FileAudio* f;

    Brano(FileAudio* p): f(p->clone()) {}
    Brano(const Brano& x): f((x.f)->clone()) {}
    Brano& operator=(const Brano& x) {
        if(this != &x) {
            delete f;
            f=(x.f)->clone();
        }
        return *this;
    }
    ~Brano() {delete f;}
    };
};
```

A SOLUZIONI DEGLI ESERCIZI

```
vector<Brano> brani;

public:
    vector<Mp3> mp3(double dim, int br) const {
        vector<Mp3> ris;
        vector<brano>::const_iterator it = brani.begin();
        Mp3* p=0;
        for( ; it<brani.end(); ++it) {
            if( (*it).f->dimensione() >= dim) {
                p=dynamic_cast<Mp3*> ((*it).f);
                if(p && p->bitrate() > br)
                    ris.push_back(*p);
            }
        }
    }

    vector<FileAudio*> braniQual() const {
        vector<FileAudio*> ris;
        vector<Brano>::const_iterator it = brani.begin();
        for( ; it<brani.end(); ++it) {
            if( (*it).f->qualita() ) {
                WAV* q = dynamic_cast<WAV*> ((*it).f);
                if(q)
                    { if(q->LossLess()) ris.push_back(q); }
                else ris.push_back((*it).f);
            }
        }
    }

    void insert(Mp3* q) {
        vector<Brano>::iterator it = brani.begin();
        bool trovato = false;
        for( ; it<brani.end() && !trovato; ++it) {
            Mp3* p = dynamic_cast<Mp3*> ((*it).f);
            if(p && *p == *q)
                trovato = true;
        }
        if(!trovato) brani.push_back(brano(q));
    }
};
```

A.2 Altri esercizi del testo

Soluzione Esercizio 2.5.1.