

CAPITOLO 1

1.1 Intro

Il C++ è un linguaggio orientato agli oggetti (**Object Oriented**) e a **tipizzazione statica** (cioè il tipo di ogni variabile è sempre esplicitamente determinato nel codice sorgente).

Un programma è costituito da un insieme di algoritmi ed un insieme di dati su cui operano gli algoritmi. Si parla di:

-Programmazione procedurale quando si lavora sugli algoritmi, definendo funzioni che permettono di estendere il linguaggio con nuove istruzioni.

Una funzione descrive un algoritmo, ma può anche contenere la definizione di alcuni tipi di dato usati dall'algoritmo;

-Programmazione ad oggetti quando si opera sui tipi di dato, definendo classi che permettono di estendere il linguaggio con nuovi tipi di dato.

Una **classe** descrive un tipo di dato e contiene le funzioni che operano su tali dati.

Caratteristiche fondamentali di un linguaggio OO legate al concetto di classe:

- Incapulamento e Occultamento dell'informazione;
- Polimorfismo (cioè che se una espressione il cui tipo è descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B che è sottoclasse di A);
- Ereditarietà.

Strumenti che permettono di integrare ed estendere i paradigmi di programmazione procedurale e ad oggetti:

- Template di funzione;
- Template di classe;
- Gestione delle eccezioni.

1.2.1 Namespace

Un problema della programmazione a moduli (programma diviso in più file) è quello dell'inquinamento dello spazio dei nomi in cui si possono provocare conflitti tra identificatori. Ad es:

<pre>Complex.h struct Complex{ }; double module(Complex);</pre>	<pre>include "Complex.h" //Altro file struct Complex{ }; //Dichiarazione illegale, dato che la struct è già inclusa in Complex.h void f(){...} //Qui si vorrebbe usare entrambi i tipi Complex ma non è possibile</pre>
----------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Con **namespace** possiamo incapsulare dei nomi che altrimenti inquinerebbero il namespace globale. Si usa quando ci si aspetta che il codice sia usato in ambienti software esterni. Es:

<pre>//Definizioni.h namespace newSpace{ struct Complex{ }; //membri del namespace newSpace double module(Complex); }</pre>	Con newSpace si identifica uno spazio dei nomi separato dal namespace globale, perciò si possono mettere dichiarazioni e definizioni qualsiasi. Esso non cambia il significato delle dichiarazioni che contiene, ne modifica soltanto la visibilità.
---------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Le dichiarazioni di un namespace non sono immediatamente visibili al programma, sono accessibili tramite l'operatore di scooping " :: ". Es:

<pre>//Lib_UNO.h namespace SPAZIO_UNO{ struct Complex{ ... }; void f(Complex c){ ... } }</pre>	<pre>//Lib_DUE.h namespace SPAZIO_DUE{ struct Complex{ ... }; void g(Complex c){ ... } }</pre>	<pre>#include "Lib_UNO.h" #include "Lib_DUE.h" void funzione(){ SPAZIO_UNO::Complex var1; SPAZIO_UNO::f(var1); SPAZIO_DUE::Complex var2; SPAZIO_DUE::g(var2); SPAZIO_DUE::g(var1); //Errore perché var1 fa parte del namespace SPAZIO_UNO }</pre>
--------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Un **alias** di un namespace permette di associare ad un namespace esistente un nome alternativo. Es:

<pre>#include "Lib_UNO.h" namespace UNO=SPAZIO_UNO; void funzione(){ UNO::Complex var1; UNO::f(var1); }</pre>

Quando l'identificazione del namespace è lunga si può usare la direttiva d'uso **using** che permette di rendere le dichiarazioni a cui si vuol fare riferimento di un namespace visibili senza qualificazione.

```
#include "Lib_UNO.h"
#include "Lib_DUE.h"

using namespace SPAZIO_UNO;

void funzione(){
    Complex var1;
    f(var1);
    SPAZIO_DUE::Complex var2;
    SPAZIO_DUE::g(var2);
}
```

-Il namespace a cui si riferisce l'using deve essere già stato dichiarato con l'include;

```
#include "Lib_UNO.h"
#include "Lib_DUE.h"

using SPAZIO_UNO::Complex;
using SPAZIO_DUE::g;

void funzione(){
    Complex var1;
    SPAZIO_UNO::f(var1);
    SPAZIO_DUE::Complex var2;
    g(var2);
}
```

La dichiarazione d'uso fornisce anche un meccanismo più selettivo che permette di rendere visibile in un namespace una singola dichiarazione.

Le librerie del C++ standard sono dichiarate nel namespace chiamato **std**. (Se non lo si inserisce il cout non compilerebbe)

```
#include <iostream>
using namespace std;
int main(){
    cout<<"Ciao"<<endl;
}
```

```
#include <iostream>
int main(){
    std::cout<<"Ciao"<<std::endl;
}
```

E' meglio evitare di usare direttive d'uso per rendere visibili i nomi del namespace std tipo using std::cout; oppure using std::endl; per il problema dell'inquinamento globale.

E' meglio usare delle dichiarazioni d'uso:

```
#include <iostream>
using std::cout;
using std::endl;

int main(){
    cout<<"Ciao"<<endl;
}
```

1.2.1 Argomenti di default

Un argomento di default è un valore dato nella dichiarazione di un parametro formale x di una funzione F() che il compilatore inserisce automaticamente quando non viene fornito alcun argomento attuale esplicito per il parametro x di una chiamata a F(). Sono utili e convenienti dato che permettono di usare un unico nome di funzione in situazioni differenti.

Es:

```
double potenza(double x, int n=2){
}
```

NB: non è possibile avere un argomento di default seguito da un argomento non di default.

```
void F(double x, int n=5, string s){ ... } //Illegale perché sting s non è di default
void G(double x, int n=5, string s="ciao"){ ... } //OK

G(3.2); //OK, x viene inizializzato, => è come se avessi G(3.2, 5, ciao);
G(); //Illegale perché x rimarrebbe indefinito
G(3,3); //OK, n viene modificato, => è come se avessi G(3,3,ciao);
G(3,3,"pippo"); //OK
```

1.2.1 Puntatori a funzione

Un puntatore a funzione contiene l'indirizzo in memoria di una data funzione.

```
#include <math.h> //contiene le funzioni sin(double) e cos(double)
#include <iostream>
using namespace std;

double F(double d) return d+3.14;

int main(){
    double (*punt)(double); //punt è un puntatore a funzione con lista dei parametri (double) e tipo di ritorno double.NB: parentesi obbligatorie!!
    pf=&sin; cout<<(*pf)(0)<<endl; //stampa 0
    pf=&cos; cout<<(*pf)(0)<<endl; //stampa 1
    pf=&F; cout<<(*pf)(0)<<endl; //stampa 3.14
}
```

1.2.1 Typedef

Permette di dichiarare un alias per un tipo già esistente. Si introduce quindi un nuovo identificatore per riferirsi ad un tipo pre-esistente. Di solito viene usato per ottenere un identificatore compatto e dal nome significativo per qualche tipo complesso.

```
typedef unsigned short int Intero;
typedef const int *PuntCostante;
typedef long double ArrayReale[20];

typedef struct{
    double parte_reale;
    double parte_immaginaria;
} Complesso; //qui "Complesso" è l'alias della struttura anonima precedente(dato che non è stato inizializzato alcun nome
              per la struttura)

typedef enum{lan,mar,mer,gio,ven,sab,dom} Giorno; //Qui "Giorno" è l'alias di un tipo enum
```

NB: E' possibile dichiarare tipi anonimi solo per struct, enum ed union.

1.2.1 Virgola

Separa le espressioni e le valuta procedendo da sinistra verso destra e ritorna il valore solo dell'ultima espressione (Raramente usato)

```
int main(){
    int a=0, b=1, c=2, d=3, e=4;
    a= (b++, c++, d++, e++); //ora ed a=e=5 b=2 c=3 d=4 e=5 (il c++ prima si assegna a=e e dopo fa e++)
}
```

1.2.1 Il tipo string

E' una classe definita nella libreria standard STL, è una istanziazione di un template classe e perciò si deve includere il file header associato.

Es:

```
string st;
st.size(); //ritorna la lunghezza di st;
st.empty(); //ritorna true se st è vuota;
-Si possono usare gli operatori: -uguaglianza ==
                                -disuguaglianza !=
                                -d'ordine <,>,<=,>=
                                -concatenazione +
posiz=st.find(y); //ritorna la posizione della prima occorrenza del caratt y su st, se non trova niente restituisce -1.
```

```
#include <string>
using namespace std;
int main(){
    string st1, st2;
    int pos;
    string st1("ecco una stringa");
    string st1="ecco una stringa"; //Equivalenti
    string st2(st1); //dichiara st2 e la inizializza come copia di st1
    getline(cin,st1); //re-inizializzo st1 con valore letto da cin ⇒ st1="FLAG: rosso bianco verde"

    pos=st1.find("rosso"); //cerca sottostringa "rosso"
    if(pos==string::npos) "allora rosso non è sottostringa di st1"
        else "pos è la posizione del carattere della prima occorrenza di rosso in st1"

    st2=st1.substr(pos,5); //ritorna la sottostringa di st1 di lunghezza 5 a partire dalla posiz pos(cioè quella di rosso) ⇒ st2="rosso"

    st1.replace(pos,5,"blu"); //sostituisce i 5 caratteri a partire dalla posizione pos con la stringa "blu" ⇒ st1="FLAG: blu bianco verde"
}
```

CAPITOLO 2 – CLASSI E OGGETTI

2.1 Tipi di dato astratti (ADT)

Un tipo di dato astratto è costituito da un insieme di valori e da un insieme di operazioni che permettono di manipolare tali valori.

La specifica di un ADT deve distinguere l'**interfaccia pubblica** dalla sua rappr interna (cioè il modo in cui i valori sono rappresentati) detta **parte privata** che non deve essere accessibile all'utente dell'ADT. Si possono quindi definire oggetti ADT che possono essere manipolati solo tramite metodi pubblici forniti dal tipo.

Su un ADT vengono definite dall'utente delle operazioni aggiuntive tramite funzioni che però non possono far uso della rappr interna del tipo. Se questa dovesse essere modificata allora sarà necessario modificare di conseguenza solo l'implementazione delle operazioni proprie mentre non ci sarà alcuna modifica per le parti del programma che usano oggetti di quel tipo.

Ad esempio il tipo **int** fornisce le operazioni di somma, moltiplicaz, uguaglianza etc che sono realizzate sfruttando la rappresentazione interna dei valori interi. Il tipo **struct** invece non rispetta il concetto di ADT, dato che la sua rappr interna è visibile all'esterno dato che qualunque funzione esterna può usare i campi dati che costituiscono la definizione di una struttura.

Es di ADT:

```
//file "complessi.h"
struct comp{
    double re,im;
};

comp iniz_compl(double,double);
double reale(comp);
double immag(comp);
comp somma(comp,comp);
```

```
//file "complessi.cpp"
#include "complessi.h"

comp iniz_compl(double re,double im){
    comp x;
    x.re=re;
    x.im=im;
    return x;
}

double reale(comp x){ return x.re; }
double immag(comp x){ return x.im; }
comp somma(comp x, comp y){
    comp z;
    z.re=x.re+y.re;
    z.im=x.im+y.im;
    return z;
}
```

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

int main(){
    comp z1;
    comp x1=iniz_compl(0.3, 3.1);
    comp y1=iniz_compl(3, 6.1);
    z1=somma(x1,y1);

    cout<<reale(z1)<<immag(z1);

    //ora usiamo la rappr interna dell'ADT

    comp x2={0.3, 3.1}, y2={3, 6.3};
    comp z2;
    z2.re=x2.re+y2.re;
    z2.im=x2.im+y2.im;

    cout<<z2.re<<z2.im;
}
```

2.2 Classi

Permettono di estendere il linguaggio con nuovi tipi di dato astratti.

Una classe si specifica in 2 parti con:

- definizione dell'interfaccia della classe**, che consiste nella dichiarazione dei campi dati e dei metodi della classe;
- dichiarazione della classe e la definizione dei suoi metodi**.

Dichiarazione della classe:

```
class orario{
public:
    //metodi della classe
    int Ore(); //selettore ore
    int Minuti(); //selettore minuti
    int Secondi(); //selettore secondi

private:
    //campo dati della classe
    int sec; // #secondi dopo la mezzanotte
};
```

Equivalente:

```
class orario{
public:
    int Ore(){ return sec/3600; }
    int Minuti(){ return (sec/60)%60; }
    int Secondi(){ return sec%60; }

private:
    int sec;
};
```

Implementazione dei metodi (o funzioni) su un file.h:

```
int orario::Ore(){ return sec/3600; }
int orario::Minuti(){ return (sec/60)%60; }
int orario::Secondi(){ return sec%60; }
```

In questo caso i metodi sono scritti **inline**. E' però opportuno tenere separate dichiarazione e definizione di una classe e normalmente esse sono memorizzate su due file distinti. (nel progetto non si usa l'inline)

E' possibile dichiarare variabili di tipo orario in ogni punto del programma in cui la dichiarazione della classe orario sia visibile.

```
int main(){
    orario mezzanotte; //oggetto della classe orario
    cout<<mezzanotte.Secondi()<<endl; //esegue il metodo Secondi() sull'oggetto mezzanotte
}
```

■ mezzanotte è l'oggetto di invocazione del metodo Secondi()

2.2.1 Puntatore *this

Quando viene dichiarato un oggetto come "mezzanotte" di tipo orario, viene riservata una zona di memoria per il valore del campo dati intero "sec", perciò ogni oggetto di tipo orario ha un proprio campo dati sec mentre in memoria c'è un'unica copia del codice oggetto dei metodi Secondi(), Minuti() ed Ore() ed è tale codice che viene eseguito quando viene effettuata la chiamata mezzanotte.Secondi().

L'implementazione dei 3 metodi Secondi() Minuti() Ore() usano il campo dati sec.

Dato che ogni oggetto ha il proprio campo dati sec allora se il metodo Secondi() viene invocato su mezzanotte.Secondi() esso deve usare il campo sec di mezzanotte, mentre se viene invocato mezzogiorno.Secondi() usa il campo sec di mezzogiorno. Ciò avviene tramite il parametro implicito **this** di tipo puntatore ad oggetti della classe stessa.

Ad esempio, per i metodi della classe orario, il parametro this è di tipo orario*.

Quando un metodo viene invocato su qualche oggetto di invocazione, al parametro implicito this è automaticamente assegnato l'indirizzo dell'oggetto(this punta all'oggetto).

Perciò: -se invochiamo mezzanotte.Secondi() ⇒ allora this punta all'oggetto di invocazione mezzanotte;
-se invochiamo mezzogiorno.Secondi() ⇒ allora this punterà a mezzogiorno.

NB: All'interno di un metodo ci si può riferire all'oggetto di invocazione tramite la dereferenziazione *this.

Es:

```
int orario::Secondi(){
    return ((*this).sec)%60;
}
```

■ L'utilizzo esplicito di this è **necessario** nella definizione dei **metodi che devono restituire l'oggetto stesso dell'invocazione**:

```
class A{
public: A f();
private: int a;
};
```



```
A A::f(){
    a=5;
    return *this;
}
```

2.2.2 Parte private e pubblica

private e **public** sono gli specificatori d'accesso. Indicano che il campo dato sec appartiene alla parte privata mentre i tre metodi alla pubblica.

Dall'esterno si può solamente accedere alla parte pubblica di una classe, i membri della parte privata sono inaccessibili.

```
orario o;
cout<<o.Ore()<<endl; ⇒ OK, Ore() è pubblico
cout<<o.sec<<endl; ⇒ Errore, sec è privato
```

- La parte privata contiene dichiarazioni dei campi dati ed eventuali metodi di utilità che il progettista della classe usa per implementare la classe ma che non fanno parte della sua interfaccia pubblica;
- La parte pubblica contiene dichiarazioni dei metodi che definiscono l'interfaccia pubblica dell'ADT che sono disponibili agli utenti della classe.
- Un membro che non sta in nessuna delle due parti apparterrà di default alla parte privata.

2.2.2 Costruttori

Servono per assegnare valori ad un oggetto. Per far sì che all'oggetto "mezzanotte" corrisponda esattamente il valore 00:00:00 dobbiamo assegnare al campo dati sec di mezzanotte l'intero 0.

```
orario mezzanotte;
mezzanotte.sec=0;
```

⇒Questo non funziona, dato che sec è un membro privato.

Per fare ciò si usano i **costruttori**, metodi con lo stesso nome della classe e senza tipo di ritorno che vengono invocati automaticamente quando viene dichiarato un oggetto. Sono **dichiarati nella parte pubblica**. (si può dichiararli nella parte privata ma sarà inutilizzabile dall'esterno⇒errore logico). Es:

```
class orario{
public:
    orario(); //costruttore senza parametri
};
```

```
orario::orario(){ //def costruttore default
    sec = 0;
}
```

```
int main(){
    orario mezzanotte; //il costruttore di default
                        //è invocato implicitamente
    cout<<mezzanotte.Ore()<<endl; //stampa 0
}
```

Si possono definire più costruttori(devono differire nella lista dei parametri). Si fa "**overloading**" dell'**identificatore** del metodo che ha lo stesso nome della classe.

```
class orario{
public:
    orario(); //costruttore default
    orario(int,int); //costruttore ore-minuti
    orario(int,int,int); //costruttore ore-minuti-sec
    int Ore(); //selettore Ore
    int Minuti(); //selettore Minuti
    int Secondi(); //selettore Secondi
private:
    int sec; //campo dati
};
```



```
orario::orario(){ //def costruttore default
    sec=0;
}
orario::orario(int ore, int min){ //def costruttore ore-minuti
    if(ore<0 || ore>23 || min<0 || min>59) sec=0;
    else sec=ore*3600+min*60;
}
orario::orario(int ore, int min, int s){ //def costruttore ore-min-sec
    if(ore<0 || ore>23 || min<0 || min>59 || s<0 || s>59) sec=0;
    else sec=ore*3600+min*60+s;
}
```

Es:

```
orario o;
o=orario(12,33,25); //il costruttore crea inizialmente un oggetto anonimo
                    //a cui non è associato nessun identificatore per poi
                    //assegnarlo all'oggetto o.
```

Es:

```

orario adesso_precio(14,25,47); //uso costr ore-min-sec
orario adesso(14,25);           //uso costr ore-min
orario mezzanotte;              //uso costr default
orario mezzanotte2;
orario troppo(27,25);           //uso costr ore-min

cout<<adesso_precio.Secondi(); //stampa 47
cout<<adesso.Minuti();         //stampa 25
cout<<mezzanotte.Ore();        //stampa 0
cout<<troppo.Ore();            //stampa 0

```

Viene invocato un costruttore anche quando si crea dinamicamente sullo heap un oggetto tramite il **new**.

```

orario* ptr=new orario;           cout<<ptr->Ore()<<endl; //stampa 0
orario* ptr1=new orario(14,25);   cout<<ptr->Ore()<<endl; //stampa 14

```

■ Il **costruttore standard** interviene quando in una classe non viene dichiarato esplicitamente nessun costruttore. Esso non ha parametri, **lascia indefiniti i valori dei campi dati dei tipi primitivi**(int, float, bool etc) **e derivati**(puntatori, riferimenti, array), **mentre per i campi dati di tipo classe richiama il corrispondente costruttore di default**.

```

class orario{
public:
    //non dichiaro nessun costruttore
};

```



orario o; //viene invocato costruttore standard di default

NB: Quando viene dichiarato almeno un costruttore, il costruttore standard non è più disponibile!

```

class orario{
public:
    orario(int,int);
};

```



orario o; //non compila dato che manca costruttore di default

Altri modi per dare valore ad oggetti orario:

```

orario adesso(11,55); //costruttore ore-minuti
orario copia;          //costruttore default
copia=adesso;          //assegnazione
orario copia1=adesso; //costruttore copia
orario copia2(adesso); //equivalente

```

-Il **costruttore di copia** crea un nuovo oggetto copia1 tramite una copia dei campi dati dell'oggetto "adesso".
 -Il costruttore di copia di una qualsiasi classe C ha segnatura C(const C&).

2.2.3 Costruttori come convertitori di tipo

```

orario::orario(int o){
    if(o<0 || o>23 ) sec=0;
    else sec=o*360;
}

```

I costruttori con un solo parametro oltre che ad essere usati normalmente come gli altri costruttori, funzionano anche come **convertitori di tipo**. Possono essere infatti usati per conversioni implicite.

Sia C(T) costruttore della classe C ad un solo parametro di tipo T ⇒ Allora il costruttore C(T) sarà automaticamente invocato ogni volta che compare un valore di tipo T dove invece ci si aspetterebbe un valore diverso di tipo C.

NB: La **conversione implicita** avviene a run-time tramite un'invocazione del costruttore ad un argomento che costruisce un oggetto temporaneo.

Con x=8; si tenta di assegnare il valore intero 8 all'oggetto x di tipo orario. Ciò provoca conversione implicita dell'intero in un oggetto di tipo orario mediante il costruttore orario(int).

```

orario x,y;
x=8; //x=orario(8);
y=8+12; //y=orario(8+12);

```

Effetti dell'assegnazione:

1. Viene invocato il costruttore orario(int) con parametro attuale 8 che crea un oggetto temporaneo anonimo;
2. L'oggetto temporaneo viene assegnato all'oggetto x;
3. L'oggetto temporaneo viene deallocato.

Tutto ciò vale anche se si definiscono costruttori con argomenti di default.

```

class orario{
public:
    orario(int=0, int=0, int=0);
    ....
};

```



```

orario::orario(int o, int m, int s){ //costruttore con 3 arg di default
    if(o<0 || o>23 || m<0 || m>59 || s<0 || s>59) sec=0;
    else sec=o*3600+m*60+s;
}

```

⇒Funziona come costruttore a 0,1,2,3 parametri ed anche come convertitore di tipo da int ad orario.

Definendo un costruttore C(T) con un solo parametro di tipo T in una classe C \Rightarrow si introduce una conversione implicita dal tipo T del parametro del costruttore al tipo C della classe.

(ok anche se def costruttori con più di un parametro dei quali il primo tipo è T, mentre per tutti i successivi parametri è previsto valore di default)

Punto sulla classe orario:

```
//file "orario.cpp"
class orario{
public:
    orario();           //costruttore default
    orario(int,int);    //costruttore ore-minuti
    orario(int,int,int); //costruttore ore-minuti-sec
    int Ore();          //selettore Ore
    int Minuti();       //selettore Minuti
    int Secondi();      //selettore Secondi
private:
    int sec;           //campo dati
};

orario::orario(int o, int m, int s){ //costruttore con 3 arg di default
    if(o<0 || o>23 || m<0 || m>59 || s<0 || s>59) sec=0;
    else sec=o*3600+m*60+s;
}

int orario::Ore(){ return sec/3600; } //metodi
int orario::Minuti(){ return (sec/60)%60; }
int orario::Secondi(){ return sec%60; }
```

```
#include <iostream>
#include "orario.cpp"
using std::cout;
using std::endl;

int main(){
    orario s;
    s=6;
    cout<<s.Ore()<<":"<<s.Minuti()<<":"<<s.Secondi()<<endl; "06:00:00"
    orario t=5+2*2; //equivale ad t=orario(5+2*2,0,0);
    cout<<t.Ore()<<":"<<t.Minuti()<<":"<<t.Secondi()<<endl; "09:00:00"
    orario r(12, 45); //equivale ad r=orario r(12, 45, 0);
    cout<<r.Ore()<<":"<<r.Minuti()<<":"<<r.Secondi()<<endl; "12:45:00"
    orario a; //equivale ad a=orario a(0,0,0); "00:00:00"
    orario b(7); //equivale ad b=orario b(7,0,0); "07:00:00"
}
```

- In qualche caso si può non volere che un costruttore ad un parametro sia richiamato implicitamente come convertitore di tipo. Basta perciò aggiungere explicit prima del costruttore ad un parametro:

```
class orario{
public:
    explicit orario(int=0);
};
```

2.4.2 Operatori espliciti di conversione

In una classe C è possibile definire una conversione implicita dal tipo C ad un altro tipo T tramite degli operatori espliciti di conversione.

```
class orario{
public:
    operator int(){ return sec; } //conversione orario $\Rightarrow$ int
};
```

```
orario o(14,37);
int x=o; //viene richiamato implicitamente l'operatore int() sull'oggetto o
```

Side Effect: quando una funzione produce un effetto collaterale, cioè quando modifica un valore o uno stato al di fuori del proprio scoping locale, ad esempio quando modifica una variabile globale o uno dei suoi argomenti.

2.5 Metodi Costanti

L'invocazione di un metodo su un oggetto di invocazione può modificare lo stato di quell'oggetto(cioè il suo campo dati). In questo caso, il metodo provoca Side Effect sull'oggetto di invocazione.

Es: i due metodi alla parte pubblica di orario:

```
class orario{
public:
    ...
    orario UnOraPiuTardi();
    void AvanzaUnOra();
private:
    int sec;           //campo dati
};
```

```
orario orario::UnOraPiuTardi(){
    orario aux;
    aux.sec=(sec+3600) % 86400;
    return aux;
}

void orario::AvanzaUnOra(){
    sec=(sec+3600) % 86400;
}
```

```
int main(){
    orario mezzanotte;
    cout<<mezzanotte.Ore();           //stampa 0
    orario adesso(15);
    cout<<adesso.Ore();               //stampa 15
    adesso=mezzanotte.UnOraPiuTardi();
    cout<<adesso.Ore();               //stampa 1
    cout<<mezzanotte.Ore();           //stampa 0
    mezzanotte.AvanzaUnOra();
    cout<<mezzanotte.Ore();           //stampa 1
}
```

In questo caso, al contrario di AvanzaUnOra, il metodo **UnOraPiuTardi()** dovrebbe essere dichiarato costante in quanto non provoca side effect all'oggetto di invocazione.

Per evitare modifiche non volute dell'oggetto di invocazione da parte di un metodo si può dichiarare quel metodo costante usando il **const** che dichiara esplicitamente che ogni invocazione di `obj.metodo()`; non provocherà side effects su `obj`.

```
class orario{
public:
    ...
    void StampaSecondi() const;
};

void orario::StampaSecondi() const{
    cout<<sec<<endl;
}
```

In questo caso il compilatore controlla che nella definizione del metodo dichiarato costante non compaia alcuna istruzione che possa alterare il valore dell'oggetto di invocazione. In caso contrario ci sarà un errore di compilazione.

Anche un oggetto può essere dichiarato costante, ciò significa che tale oggetto non può essere modificato dopo che è stato costruito.

NB: tutti i campi di un oggetto costante diventano costanti.

```
const orario LE_DUE(14);
const orario LE_TRE(15);
LE_DUE=LE_TRE; //Errore, tento di modificare un oggetto costante.
LE_DUE.sec=50400; //Errore, tento di modificare il campo di un oggetto costante
```

NB: l'oggetto di invocazione di un metodo costante diventa costante. Infatti nel corpo di un metodo costante di classe C, il puntatore `this` all'oggetto di invocazione ha tipo **const C*** invece che **C*** e di conseguenza l'oggetto di invocazione `*this` ha tipo **const C**.

NB: un oggetto costante si può usare come oggetto di invocazione solo per metodi dichiarati costanti. Se si tenta di invocare un metodo non costante su un oggetto costante il compilatore dichiara errore.

```
const orario LE_TRE(15);
LE_TRE.StampaSecondi(); //Stampa 15
orario o;
o=LE_TRE.UnOraPiuTardi(); //Errore, perché UnOraPiuTardi() anche se non modifica l'oggetto di invocazione non è stato dichiarato metodo costante
```

NB: Se un metodo `m()` non ha side effect sull'oggetto di invocazione, è obbligatorio dichiarare il metodo `m()` costante. Ciò permette di invocarlo anche su oggetti costanti (se `m()` non fosse dichiarato costante ciò non sarebbe possibile)

NB: I costruttori sono dei metodi non costanti che però possono venire invocati su oggetti dichiarati costanti. (Se così non fosse non sarebbe ovviamente possibile costruire oggetti costanti)

2.6 Campi dati e metodi statici

Finora i metodi della classe `orario` sono stati pensati per operare su di uno specifico oggetto di invocazione. Potremmo avere l'esigenza che la classe `orario` ci fornisca un valore OraDiPranzo di tipo orario che sia sempre lo stesso, perciò una specie di costante della classe `orario`.

Per ottenere ciò si può definire un metodo costante OraDiPranzo() che ritorna un oggetto della classe `orario`.

```
class orario{
public:
    orario OraDiPranzo() const;
};

orario orario::OraDiPranzo() const{ return orario(13,14); }
```

Questo è però un metodo mal definito. Infatti per stampare l'ora di pranzo è necessario avere qualche oggetto di tipo `orario` da usare come oggetto di invocazione, anche se tale oggetto non sarà usato dal metodo `OraDiPranzo()`;

Si possono associare sia metodi che campi dati all'intera classe invece che ai singoli oggetti della classe tramite il metodo **static** che perciò **si usa quando l'azione del metodo è indipendente dall'oggetto di invocazione** (quindi quando l'oggetto di invocazione non è necessario per definire il metodo). Può essere sia pubblico che privato. Ciò permette di invocare un metodo statico senza oggetto di invocazione.

```
class orario{
public:
    static orario OraDiPranzo();
};

orario orario::OraDiPranzo(){ return orario(13,14); }

cout<<"Si pranza alle "<<orario::OraDiPranzo().Ore();
cout<<" e "<<orario::OraDiPranzo().Minuti()<<" minuti";
```

NB: Const non ha senso per un metodo statico dato che `OraDiPranzo()` non ha oggetto di invocazione.

NB: I metodi static non hanno il parametro implicito `this`. All'esterno della classe si invocano tramite l'operatore di scoping.

NB: I campi dichiarati static si devono inizializzare all'esterno della classe.

La memoria per i campi dati statici è unica per tutti gli oggetti ed è allocata una volta per tutti.

Es:

```
class C{ ⇒ contare oggetti istanziati
public:
    C(int); //costruttore ad un argomento
    static int count; //campo dati statico pubblico
private:
    int dato;
};
int C::count=0; //inizializ campo dati statico
C::C(int n){ count++; dato=n; }
```



```
int main(){
    C c1(1), c2(2);
    cout<<C::count; //stampa 2
}
```


2.7 Overloading degli Operatori

Vogliamo aggiungere alla classe orario un'operazione che permette di sommare due oggetti di tipo orario:

```
orario orario::Somma(orario o){
    orario aux;
    aux.sec=(sec + o.sec) % 86400; //NB:accedo a parametro privato di o
    return aux;
}
```



```
orario ora(22,45);
orario DUE_ORE_E_UN_QUARTO(2,15);
ora=ora.Somma(DUE_ORE_E_UN_QUARTO)
```

Un modo più elegante per farlo è definire un overloading, cioè una ridefinizione, dell'operatore somma +.

Ciò si ottiene definendo una funzione (anche esterna alla classe) il cui identificatore è **operatorOP** in cui ogni occorrenza dell'operatore OP equivale ad un'invocazione di tale funzione.

```
class orario{
public:
    orario operator+(orario);
    ...
};
//overloading operatore:
orario orario::operator+(orario o){
    orario aux;
    aux.sec=( sec + o.sec ) % 86400;
    return aux;
}
```



```
orario ora((22,45);
orario DUE_ORE_E_UN_QUARTO(2,15);
ora=ora+DUE_ORE_E_UN_QUARTO;
ora=ora+3; //ok, ma solo se l'oggetto sta a sx (ora=3+ora; non compila)
```

Operatori Sovraccaricabili:

+ - * / % == != < > <= >= ++ -- << >> = -> [] () & new delete

Non Sovraccaricabili:

. sizeof :: ?: typeid

La ridefinizione di un operatore binario corrisponde ad un metodo che ha un solo argomento esplicito oppure ad una funzione esterna con due argomenti, mentre un operatore unario corrisponde ad un metodo proprio senza argomenti espliciti.

- NB:**
1. Non si possono cambiare le proprietà sintattiche dell'operatore, ovvero posizione (prefissa, infissa, postfissa), numero di operandi, precedenza e associatività;
 2. Fra gli argomenti dell'operatore ridefinito deve esserci almeno un argomento di un tipo definito dall'utente;
 3. Gli operatori = (assegnazione), [] (indicizzazione), () (chiamata a funzione), -> (selezione di membro tramite puntatore) possono essere definiti solo come metodi propri (così si fa in modo che il primo operando sia un L-valore modificabile).

2.7.1 Assegnazione Standard

Se a e b sono due oggetti della stessa classe C e non è stato ridefinito l'operatore di assegnazione per C, allora l'istruzione a=b; ha come effetto quello di assegnare ad ognuno dei campi dati di a il valore del corrispondente campo dati di b. L'assegnazione standard è quindi automaticamente invocata nel caso in cui il programmatore non ridefinisca esplicitamente l'operatore di assegnazione =.

La segnatura standard per una generica classe C è

```
C& operator = (const C&);
```

2.7.2 Costruttore di Copia Standard

E' una funzione che ha un comportamento simile a quella dell'assegnazione standard. Ha segnatura

```
C(const C&);
```

 e viene invocato automaticamente:

1. Quando un oggetto viene dichiarato ed inizializzato con un altro oggetto della stessa classe :

```
orario adesso(14,30);
orario copia = adesso; //dichiaro e inicializzo copia
orario copia1=adesso; //analogo
```

2. Quando un oggetto viene passato per valore come parametro attuale in una chiamata di funzione :

```
ora= ora.Somma(DUE_ORE_E_UN_QUARTO);
```

3. Quando una funzione ritorna per valore un oggetto tramite il return :

```
return aux;
```

⇒ In tutti e tre i casi viene costruito un nuovo oggetto di tipo orario i cui campi dati sono inizializzati con i valori corrispondenti ai campi dati dell'oggetto da cui è stato copiato, cioè di adesso, DUE_ORE_E_UN_QUARTO ed aux.

Assegnazione standard vs Costruttore di copia:

- Il costruttore di copia crea un nuovo oggetto inizializzandolo da un altro oggetto dello stesso tipo;
- l'assegnazione cambia il valore ad un oggetto già esistente.


2.7.3 Overloading di operatori con funzioni esterne

Come detto è possibile sovraccaricare degli operatori come funzioni esterne alla classe e non come metodi propri della classe.

Vogliamo definire funzione esterna che stampa sullo stream di cout un oggetto di tipo orario.

L'operatore che permette di stampare sul cout è il << ed è tra quelli ridefinibili.

```
ostream& operator<<(ostream& os, const orario& o){           //per poter ripetere le invocazioni dell'operatore << in un'unica istruzione è necessario
    return os<<o.Ore()<<":"<<o.Minuti()<<":"<<o.Secondi();    //ritornare lo stream per riferimento
}
```



```
orario le_tre(15,0), le_due(14,0), dodici(12,0);
cout<<"adesso sono le"<<le_tre;           //stampa 15:0:0
cout<<le_tre<<"vengono prima delle"<<le_due;
cout<<"tra dodici ore saranno le"<<le_tre+dodici;    //stampa 3:0:0
```


Nell'esempio precedente con

```
orario orario::operator+(orario);
```

non era possibile eseguire $x=4+y$; in quanto il metodo operator+ può essere invocato solo se l'operando sinistro è un oggetto esplicito della classe orario.

```
orario operator+(orario x, orario y){
    int sec=x.Secondi()+y.Secondi();
    sec=sec % 60;
    int min=x.Minuti()+y.Secondi();
    min=min % 60;
    int ore=x.Ore()+y.Ore();
    ore=ore % 24;
    orario aux(ore,min,sec);
    return aux;
}
```

Se invece dichiariamo operator+ come funzione esterna, è possibile eseguire anche:



```
orario y(12,15), x;
x=y+4;
x=4+y;
x=4+5; //ok, fa la somma tra interi e poi la
        conversione del risultato
```

2.8 Incapsulamento e Modularizzazione

Nella classe orario l'ora del giorno è rappresentata internamente tramite un campo dato privati sec di tipo int.

In alternativa si sarebbero potuti usare 3 campi int per memorizzare separatamente ore, minuti e secondi.


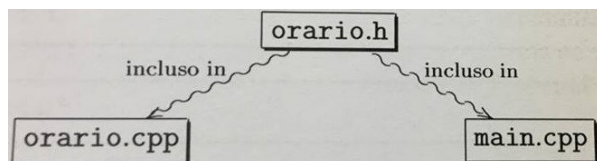
Le due scelte presentano vantaggi e svantaggi, ad esempio la seconda semplifica la realizzazione dei metodi Ore(), Minuti(), Secondi() ma complica l'implementazione di operator<<.

Qualunque sia la scelta di rappresentazione interna, essa non ha alcuna conseguenza per un utente della classe orario in quanto non può accedere alla parte privata della classe.

La modularizzazione dei programmi è un modo per nascondere all'utente esterno la dichiarazione della parte privata della classe (che altrimenti resterebbe visibile nella dichiarazione della classe) tramite la dichiarazione di una classe e dei suoi metodi in due file separati.

La dichiarazione di una classe va messa in un file header con estensione **“.h”** mentre la definizione dei metodi va in un file **“.cpp”**

E' quindi possibile mettere la definizione dei metodi Ora(), Minuti(), Secondi() sul file orario.cpp, compilarlo producendo di conseguenza orario.o contenente il codice oggetto.o e fornire agli utilizzatori della classe soltanto il file orario.o



orario.h dovrà essere incluso sia in orario.cpp che in main.cpp insieme ad eventuali file header di libreria (ad es iostream o string) tramite #include)

(L'uso nell'#include di **“ ”** significa che orario.h va cercato nella directory corrente, mentre **< >** ad es di iostream va cercato in un insieme standard di directory del compilatore)

2.8.1 Il preprocessore

E' una parte del compilatore che esegue una elaborazione preliminare del testo sorgente del programma, prima che avvenga la compilazione in codice oggetto. Esso può inserire contenuto di altri file, espandere i simboli definiti del programma e includere o escludere parti di codice dal testo che sarà effettivamente compilato. Le direttive standard sono:

```
#include #error #if #else #elif #endif
#define #ifdef #ifndef #undef #line #pragma
```

#include indica che un determinato file dovrà essere incluso nell'unità di compilazione.

#define precede la definizione di una **macro**, un simbolo seguito da una lista di argomenti formali (delimitati tra parentesi tonde) che prima della compilazione viene sostituito da una sequenza di elementi lessicali corrispondenti alla definizione della macro. Se la macro ha argomenti formali, al momento della sostituzione questi vengono rimpiazzati dagli argomenti attuali corrispondenti.

```
#define COST 123
#define max(A,B) ((A)>(B) ? (A) : (B)) //Se A>B allora è A, altrimenti B

int main(){
    x=COST;
    y=max(4,z+2);
}
```

visto dal preprocessore come:

```
int main(){
    x=123;
    y=((4)>(z+2) ? (4) : (z+2));
}
```

Se in una macro manca la definizione:

```
#define ORARIO_H
allora ad ogni occorrenza della macro viene sostituita con un
carattere spazio.
```

La definizione di una macro M ha effetto fino alla fine dell'unità di compilazione(cioè fino a fine file) oppure fino ad una delimitazione **#undef** M.

Compilando contemporaneamente più file può succedere che un file header venga incluso più volte in un file provocando un errore di compilazione, in quanto il compilatore incontra definizioni multiple di uno stesso identificatore.

Es:

```
//file "C.h"
class C{
public int x;
int x;
C(int k=4){ x=k; }
};
```

```
//file "D.h"
#include <iostream>
#include "C.h"
class D{
public:
int x;
D(int k=6){ x=k; }
void print(C c) const{ std::cout<<x+c.x; }
};
```

```
//file "main.cpp"
#include "C.h"
#include "D.h" //Non compila, perché la classe C è inclusa 2 volte
//una sul file attuale "main.cpp" ed una su "D.h"

int main(){
    C c(3);
    D d(4);
    d.print(c);
}
```

Per ovviare a questo si inseriscono nei file header delle direttive dette di **compilazione condizionale**(o di isolamento), che permettono di includere o escludere dalla compilazione segmenti di codice.

```
#ifdef name //se definito
text //esegui
#endif
```

```
#ifndef name //se non definito
text //esegui
#endif
```

```
//file "orario.h"
#ifndef ORARIO_H //se non definito
#define ORARIO_H //definisci
class orario{
.....
};
#endif
```

2.8.2 Compilazione e Linking

I due file orario.cpp e main.cpp possono essere compilati separatamente.

```
g++ -c orario.cpp //compila orario.cpp e crea file oggetto
```

La compilazione separata permette di non fornire agli utilizzatori della classe orario il file sorgente orario.cpp, evitando di rendere disponibili le definizioni dei metodi della classe. All'utilizzatore saranno forniti:

1. il file header orario.h da includere in ogni modulo utente M della classe orario;
2. il file oggetto orario.o che verrà utilizzato dal linker per generare codice eseguibile a partire dal codice oggetto.

Se voglio linkare il main.cpp e orario.o

```
g++ -c main.cpp //compila main.cpp in main.o
g++ main.o orario.o //linka il main.o con orario.o e produce l'eseguibile
```

2.8.3 Compilazione e Linking

Quando si scrivono programmi composti da diversi file sorgente, si deve far fronte al problema della ripetuta ricompilazione dei sorgenti e della successiva fase di link dei file oggetto per poter poi generare l'eseguibile. Conviene perciò individuare solo i file che sono stati modificati dall'ultima compilazione e ricompilare solo questi ultimi.

Su Linux esiste il comando **make** che serve a determinare automaticamente quali file che compongono un programma hanno bisogno di essere ricompilati. Questo comando è in grado di invocare qualunque comando di sistema che possa essere eseguito da shell (file tex, db etc).

Per utilizzarlo è necessario creare un file di nome **Makefile**, il quale descrive le relazioni esistenti tra i vari file e i comandi da eseguire per aggiornare ognuno di questi. In un programma modularizzato contenente molti file, l'eseguibile viene aggiornato dal linker usando i file oggetto, mentre i file oggetto vengono generati dal compilatore usando i file sorgente.

Ha la sintassi:

TARGET : DIPENDENZE	//TARGET= nome file eseguibile o file oggetto da ricompilare
COMANDI	//DIPENDENZE= contiene tutti i file o le azioni da cui dipende il TARGET

Una volta definito Makefile ⇒ Si invoca il comando **make** sulla shell ⇒ verranno ricompilati solo i file sorgente modificati.

Quando make andrà a ricompilare il programma:

- Se un file header.h viene modificato ⇒ allora ogni file sorgente.cpp che includeva il file header verrà ricompilato;
- Se almeno un file, che sia header o sorgente, viene modificato (quindi almeno un file oggetto è stato rigenerato)
⇒ allora tutti i file oggetto, vecchi e nuovi, verranno linkati assieme per generare l'eseguibile aggiornato.

Sia un programma C++ costituito da 5 file sorgente .cpp e 3 file header .h

Il Makefile è:

```
//CC=il comando che invoca il compilatore g++
//CFLAGS=flags per il compilatore

CFLAGS=-Wall -march=x86-64

prog_mio : main kbd video print
    &(CC) &(CFLAGS) -o prog_mio main.o kbd.o      (Il // serve per dividere un unico comando in più righe)
    //video.o help.o print.o

main : main.cpp config.h
    &(CC) &(CFLAGS) -c main.cpp -o main.o
kbd : kdb.cpp config.h
    &(CC) &(CFLAGS) -c kdb.cpp -o kdb.o
video : video.cpp config.h defs.h
    &(CC) &(CFLAGS) -c video.cpp -o video.o
help : help.cpp help.h
    &(CC) &(CFLAGS) -c help.cpp -o video.o
print : print.cpp
    g++ -c print.cpp -o print.o

clean :
    rm *.o
    echo "pulizia completata"
```

2.8.4 Modularizzazione delle classi

La definizione di classi permette di aggiungere nuovi tipi di dato ed una volta definite possono essere **usate modularmente per creare altre classi più complesse**.

Esempio: creo classe telefonata che utilizza la classe orario. Una telefonata è rappresentata da ora inizio,ora fine,numero chiamato.

```
//file "telefonata.h" definisco la dichiarazione della classe
#ifndef TELEFONATA_H
#define TELEFONATA_H
#include <iostream>
#include "orario.h"

using std::ostream;

class telefonata{
public:
    telefonata(orario,orario,int);
    telefonata();
    orario inizio() const;
    orario fine() const;
    int Numero() const;
    bool operator==(const telefonata&) const;
private:
    orario inizio, fine; //relazione has-a
    int numero;
};

std::ostream& operator<<(ostream&, const telefonata);
#endif
```

Ora bisogna aggiornare il comportamento dei costruttori in presenza di campi dati il cui tipo è una classe.

Sia una classe C con campi dati x1,x2...xk di qualsiasi tipo. L'ordine dei campi dati è determinato dall'ordine in cui essi appaiono nella definizione della classe C.

Se definiamo una **classe con costruttore generico C(tipo1,tipo2,...,tipokn){ ... }** il **comportamento del costruttore** è:

1. Per ogni campo dati xi di tipo Ti non classe(cioè di tipo primitivo) viene allocato un corrispondente spazio in memoria per contenere un valore di tipo Ti ed il valore è lasciato indefinito;
2. Ogni campo dati xi di tipo classe Ti viene costruito mediante un invocazione del costruttore di default Ti();
3. Viene eseguito il codice del corpo del costruttore.

```
//file "telefonata.cpp" definisco i metodi
#include "telefonata.h"

telefonata::telefonata(orario i, orario f, int n){
    inizio=i;
    fine=f;
    numero=n;
}

telefonata::telefonata(){
    numero=0; //gli altri campi inizio e fine sono inizializzati tramite il costruttore di default di orario
}

orario telefonata::Inizio() const{ return inizio; }
orario telefonata::Fine() const{ return fine; }
int telefonata::Numero() const{ return numero; }

bool telefonata::operator==(const telefonata & t){
    return inizio==t.inizio && fine==t.fine && numero==t.numero;
}

ostream& operator<<(ostream& s, const telefonata& t){
    return s<<"INIZIO :"<<t.Inizio()<<"Fine :"<<t.Fine()<<" Numero Chiamato: "<<t.Numero();
}
}
```

2.9 Campi dati costanti

A volte è utile dichiarare alcuni campi dati di una classe come costanti. Ad esempio una volta costruito un oggetto della classe telefonata vogliamo che il suo campo dati numero non possa venire modificato. Per inizializzare dei campi dati costanti è necessario richiamare esplicitamente il costruttore di copia:

```
class telefonata{
...
private:
    orario inizio,fine;
    const int numero;
}
```



```
telefonata::telefonata() : numero(0){ }
```

```
telefonata::telefonata(){
    numero=0; //Errore, non è possibile utilizzare
}
           l'istruzione di assegnazione per
           inizializzare dei campi costanti
```

E' anche possibile richiamare il costruttore di copia per qualsiasi campo dati, anche per quelli non costanti:

```
telefonata::telefonata(orario i, orario f, int n) : inizio(i), fine(f), numero(n) { }
```

2.10 Liste di inizializzazione dei costruttori

Abbiamo visto come un costruttore possa richiamare i costruttori di copia per creare ed inizializzare i campi dati. Un costruttore può quindi creare ed inizializzare i suoi dati tramite un'esplicita lista di inizializzazione, che consiste in una lista di invocazioni a costruttori.

Sia una classe C con lista ordinata di campi dati $x_1..x_k \Rightarrow$ Un costruttore con lista di inizializzazione per i campi dati $x_1..x_k$ è:

```
C(T1...,Tn) : x1(...) , x2(..) , .. , xn(..){ //codice }
```

Il **comportamento** del costruttore è il seguente:

1. Per ogni campo dati x_i viene ordinatamente richiamato un costruttore:
 - esplicitamente, tramite una chiamata ad un costruttore $x_i(..)$ definita nella lista di inizializzazione;
 - implicitamente, tramite una chiamata al costruttore di default $x_i()$;
2. Viene eseguito il codice del costruttore.

Campi dati di tipo riferimento: anche se non molto comune, è possibile dichiarare un campo dati c di tipo $T\&$ (cioè un alias) dove T è un qualsiasi altro tipo. Questo campo dati C deve essere completamente inizializzato tramite una chiamata al costruttore di copia inclusa nella lista di inizializzazione.

CAPITOLO 3 – CLASSI CONTAINER

Un oggetto di una classe container (o contenitore) rappresenta una collezione di elementi gestita tramite varie funzionalità, tra le quali l'inserimento e la rimozione. Di solito queste classi contengono fra i loro campi dati dei puntatori a strutture dati dinamiche ricorsive, come liste o alberi (vector, list, map).

Es: classe bolletta che rappresenta una serie di telefonate memorizzate in una lista. Prevede la possibilità di aggiungere o togliere una telefonata, controllare se la bolletta è vuota, stampare elenco telefonate, calcolare costo totale bolletta.

```
//file "bolletta.h"
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include "telefonata.h"
```

```
class bolletta{
public:
    bolletta() : first(0) { } //definizione inline
    bool Vuota() const;
    void Aggiungi_Telefonata(telefonata);
    void Togli_Telefonata(telefonata);
    telefonata Estrai();
private:
    class nodo{          //classe interna e privata
    public:
        nodo();
        nodo(const telefonata&, nodo*);
        telefonata info;
        nodo* text;
    };
    nodo* first;
};
#endif
```



```
//file "bolletta.cpp"
#include "bolletta.h"
```

```
bolletta :: nodo :: nodo() : next(0) { } //qui è implicito il costruttore di default per il campo info del nodo
```

```
bolletta :: nodo :: nodo(const telefonata& t, nodo* s) : info(t), next(s) { }
```

```
bool bolletta :: Vuota() const{ return first==0; }
```

```
void bolletta :: Aggiungi_Telefonata(telefonata t){
    first=new nodo(t,first); //aggiunge la telefonata in testa alla lista
}
```

```
void bolletta :: Togli_Telefonata(telefonata t){
    nodo* p= first;          //p punta all'inizio della lista
    nodo* prec=0;
    while( p && !(p->info==t) ){    //scorre la lista finchè non è finita o finchè non ha trovato la telefonata
        prec=p;
        p=p->next;
    }
    if(p){ if(!prec) first=p->next; //se ho trovato la telefonata la elimino
           else prec->next=p->next;
           delete p;
    }
}
```

```
telefonata bolletta :: Estrai_Una(){
    nodo* p=first;
    first=first->next;
    telefonata aux=p->info;
    delete p;
    return aux;
}
```

3.1 Classi Annidate

Nella definizione della classe bolletta, c'è una classe interna "nodo" i cui oggetti rappresentano i nodi della lista dinamica che rappresenta una bolletta.

Una classe A definita internamente ad un'altra classe C viene detta annidata o interna.

Una classe interna ad A può essere sia pubblica che privata.

-Se A è pubblica allora posso definire gli oggetti di A esternamente a C, usando l'operatore di scooping. (ad es con C::A obj;)

-Se A è privata allora il tipo di A è inaccessibile all'esterno di C. In particolare, in A posso usare solamente i membri statici della classe contenitrice C.

NB: I membri della classe annidata A in C non possono accedere ai membri di C e viceversa.

Es:

```
int x; //variabile globale

class C{
public:
    int x;
    static int s;
    class A{
    void f(int i){
        int x=sizeof(x); //NO, si riferisce alla variabile x dichiarata in C
        x=i;              //come sopra
        s=i;
        ::x=i; //OK, si riferisce all'x della variabile globale
    }
    void g(C* p, int i){
        p->x=i;
    }
};
```

3.2 Problemi dell'interferenza

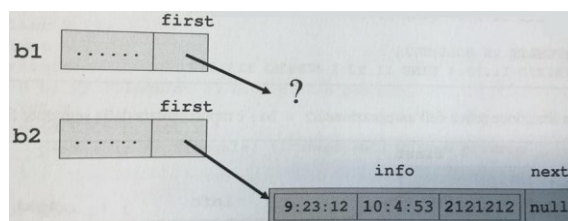
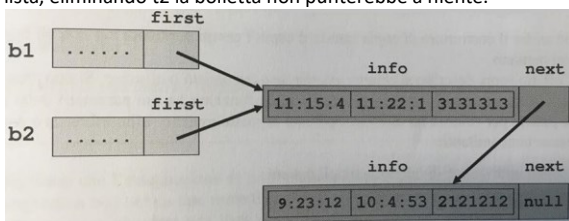
Alcuni metodi di bolletta come `Aggiungi_Telefonata()` modificano l'oggetto di invocazione e in particolari circostanze questo può provocare inconvenienti.

Es: supponiamo di avere una funzione esterna che ridefinisce l'operatore di output `<<` in modo che permetta di stampare la lista delle telefonate.

```
int main(){
    bolletta b1;
    telefonata t1(orario(9,23,12), orario(10,4,53), 212121212);
    telefonata t2(orario(11,15,4), orario(11,22,1), 313131313);

    b1.Aggiungi_Telefonata(t1);
    b1.Aggiungi_Telefonata(t2);
    cout<<b1;          //stampa b1 composta dalle due telefonate t1 e t2
    bolletta b2;
    b2=b1;
    b2.Togli_Telefonata(t1); //Problema, elimina sia t1 di b1 che di b2
    cout<<b1<<b2;        //stampa le due bollette ma solo con t1
}
```

Problema peggiore se avessimo rimosso dalla bolletta la telefonata t2, perché dato che con `Aggiungi_Telefonata()` si aggiunge la telefonata a inizio lista, eliminando t2 la bolletta non punterebbe a niente.



Ciò succede perché l'assegnazione standard copia i campi puntatore e non gli oggetti a cui essi puntano.

L'effetto descritto si chiama **interferenza tra oggetti o aliasing**. E' dovuta da:

- Condivisione di memoria tra gli oggetti;
- Funzioni che modificano oggetti.

3.3 Copie Profonde

Poiché non è possibile evitare che ci siano funzioni che modificano gli oggetti, bisogna perciò evitare o gestire opportunamente la condivisione di memoria ridefinendo sia l'assegnazione che il costruttore di copia in modo tale che essi effettuino una **copia profonda**, cioè che ogni volta che copiamo un puntatore, copiamo anche l'oggetto puntato.

Questo può però risultare molto costoso sul tempo di esecuzione(ad es lista con milioni di elementi).

3.3.1 Assegnazione Profonda

L'operatore di assegnazione `=`, che è binario ed infisso, si può ridefinire come ogni altri operatore dichiarando un metodo **operator=**.

Bisogna perciò decidere il tipo dei parametri e il tipo restituito nella dichiarazione di `operator=`.

Data l'assegnazione `x=y`; dove x è un oggetto di qualche classe C.

Supponiamo che l'assegnazione sia ridefinita internamente alla classe C come metodo, perciò \Rightarrow

- Il primo operando x viene assunto come oggetto di invocazione dell'assegnazione;
- Il secondo operando invece è un parametro formale di tipo C. Questo parametro sarà dichiarato costante e per riferimento in modo da evitare che venga effettuata inutilmente una copia del secondo operando y.

Perciò `x=y`; restituisce l'L-valore di x (che permette di usare anche `x=y=z`;

La dichiarazione da mettere nella parte pubblica per ridefinire `operator=` sarà

`C& operator=(const C&);`

Per far sì che l'assegnazione di bolletta effettui copie profonde, verranno aggiunti:

```
class bolletta{
public:
...
bolletta& operator=(const bolletta&);
bolletta(const bolletta&); //ridefinizione costr di copia profonda
private:
...
// Vengono aggiunti due metodi statici privati:
static nodo* copia(nodo*); //data lista L restituisce copia profonda di L;
static void distruggi(nodo*); //dealloca una data lista.
};
```



```
//su "bolletta.cpp"
bolletta& bolletta :: operator=(const bolletta& b){ //definizione di operator=
if(this != &b){ //obbligatorio! ESAME!!
    distruggi first;
    first=copia(b.first);
}
return *this;
}

bolletta :: bolletta(const bolletta& b) : first(copia(b.first)) { } //costruttore di copia
bolletta :: nodo* bolletta :: copia(nodo* p){
if(!p) return 0;
nodo* primo=new nodo; //invocazione del costruttore di default al nodo
primo->info=p->info; //primo punta al primo nodo della copia della lista
nodo* q=primo; //q punta al nodo temporaneo con cui scorriamo la lista
while(p->next){
    q->next=new nodo;
    p=p->next;
    q=q->next;
    q->info=p->info;
}
q->next=0;
return primo;
}

void bolletta::distruggi(nodo* p){
nodo* q;
while(p){ //scorro la lista deallocando i nodi
    q=p;
    p=p->next;
    delete q;
}
}

//soluzione alternativa ricorsiva
bolletta :: nodo* bolletta :: copia(nodo* p){
if(!p) return 0;
else return new nodo(p->info, copia(p->next));
}

void bolletta :: distruggi(nodo* p){
if(p){
    distruggi(p->next);
    delete p;
} }
}
```

(Parametri attuali=valori del main,
Parametri formali=valori dentro la funzione)

3.4 Oggetti come parametri di funzione

Ogni volta che una funzione F() viene invocata, i suoi parametri formali vengono allocati in memoria e inizializzati con i corrispondenti parametri attuali. Due meccanismi di passaggio parametri:

Passaggio per valore: i parametri formali vengono inizializzati tramite delle copie degli R-valori dei parametri attuali. Ad esempio data la funzione **T1 F(T x)** ed una espressione y con tipo T del parametro \Rightarrow con F(y), il parametro formale x viene inizializzato con il valore di y.

-Se T è un tipo classe, l'inizializzazione del parametro x comporta la chiamata del costruttore di copia della classe T.

(NB: -è molto probabile che un oggetto occupi molto spazio in memoria e quindi è **molto costoso** passarlo per valore;
-inoltre le eventuali **modifiche fatte sui parametri formali non si riflettono sui parametri attuali.**)

Passaggio per riferimento:

T1 fun(T& x) \Rightarrow x è un riferimento ad una variabile di tipo T \Rightarrow con F(y); il parametro attuale y deve essere un'espressione indirizzabile di tipo T(cioè y deve avere L-valore).

-In questo caso, il parametro formale **x diventa un alias del parametro attuale y**, senza creare alcuna copia locale alla funzione.

Perciò y può essere: -una variabile di tipo T;

-una variabile di tipo T& ritornato per riferimento da qualche funzione.

Questo meccanismo permette alle funzioni di **modificare i valori delle variabili passate per riferimento**, ed è utile quando vengono passati oggetti che occupano molto spazio.

Es: date le funzioni **T1 F(int &x)** e **int g()** \Rightarrow dato che x di F è di tipo riferimento \Rightarrow F(2) e F(g()) provocano errore di compilazione, dato che i parametri attuali non sono indirizzabili;

Quando una funzione non modifica il parametro attuale, è buona norma dichiarare costante il parametro formale, indicando quindi che la funzione non modifica il valore dell'argomento. Questo perciò vale anche quando passiamo un indirizzo, dato che questo non verrà modificato \Rightarrow Per passare un oggetto per riferimento si userà:

T1 F(const T& x)

Es: date le funzioni **T1 F(const int& x)** e **int g()** \Rightarrow F(2) e F(g()) ora risultano corrette, dato che il tipo const T& per il parametro formale permette alla funzione di essere invocata con un qualsiasi valore di T(anche non indirizzabile).

NB: se un riferimento costante viene inizializzato con un R-valore non indirizzabile(cioè senza L-valore) di un qualche tipo classe C \Rightarrow allora il costruttore di copia di C deve essere accessibile.

```
class C{
public:
    C(){ ... }
private:
    C(const C& y){ ... } //costruttore di copia privato
};
```



```
C fun1(){ ... } //ritorna C per valore

void fun2( const C& y){ ... } //parametro è c riferimento costante

void fun3(void){
    fun2(C()); //errore, costruttore di copia C inaccessibile
    fun2(fun1()); //analogo
    C y;
    fun2(y); //OK, y ha un L-valore
}
```

Es: vogliamo scrivere una funzione `Somma_Durate` esterna alla classe `bolletta` che data una bolletta `b` restituisce la somma delle durate delle telefonate della bolletta stessa.

```
orario Somma_Durate(bolletta b){ //parametro di tipo bolletta passato per valore
    orario durata; // costruttore di default di orario
    while(!b.Vuota()){
        telefonata t=b.Estrai_Una(); //estrae dal primo nodo della lista finchè non è vuota
        durata=durata+t.Fine() - t.Inizio();
    } //durata deve essere < 24 ore
    return durata;
};
//non provoca effetti sulla bolletta del main, dato che l'oggetto b è passato per valore
```

```
//su "main.cpp"
int main(){
    bolletta b1;
    telefonata t1(orario(9,23,12), orario(19,4,53), 2121212);
    telefonata t2(orario(11,15,4), orario(11,22,1), 3131313);
    b1.Aggiungi_Telefonata(t2);
    b1.Aggiungi_Telefonata(t1);
    cout<<b1; //stampa le due telefonate
    cout<<Somma_Durate(b1)<<endl; //stampa 0:38:38
    cout<<b1; //stampa sempre le due telefonate di b1
}
```

NB: `Estrai_Una` provoca effetti collaterali sulla bolletta di invocazione ma dato che abbiamo passato l'oggetto `b1` per valore e non per riferimento, tramite la chiamata `Somma_Durate(b1)` abbiamo eseguito una chiamata al costruttore di copia di bolletta che ha quindi costruito il parametro formale dentro alla funzione.
In questo modo abbiamo fatto una copia profonda, e dato che ogni volta che sommiamo una durata estraiamo una telefonata, a fine funzione il parametro formale `b` sarà completamente deallocato mentre il valore di `b1` resterà invariato.

Es: funzione esterna `Chiamate_A` che rimuove dal parametro `b` di tipo bolletta tutte le telefonate di un determinato num, restituendo una nuova bolletta contenente le telefonate rimosse.

```
bolletta Chiamate_A(int num, bolletta &b){ //parametro di tipo bolletta passato per riferimento
    bolletta selezionate,resto; //oggetti locali
    while(!b.Vuota()){
        telefonata t=b.Estrai_Una();
        if(t.Numero()==num) selezionate.Aggiungi_Telefonata(t);
        else resto.Aggiungi_Telefonata(t);
    }
    b=resto; //utilizza l'overloading di operator= di bolletta
    return selezionate;
};
```

```
int main(){
    bolletta b1;
    telefonata t1(orario(9,23,12), orario(10,4,53), 2121212);
    telefonata t2(orario(11,15,4), orario(11,22,1), 3131313);
    telefonata t2(orario(12,17,5), orario(12,22,8), 2121212);
    telefonata t4(orario(13,46,5), orario(14,0,33), 3131313);
    b1.Aggiungi_Telefonata(t4), b2.Aggiungi_Telefonata(t3)...
    cout<<b1; //Stampa tutte e 4 le telefonate di b1
    //ora elimino da b1 tutte le chiamate da 21212
    bolletta b2=Chiamate_A(2121212,b1);
    //tramite il costruttore di copia assegna a b2 le telefonate
    rimosse da b1
    cout<<b1; //stampa 2 chiamate da 3131313
    cout<<b2; //stampa 2 chiamate provenienti da 21212
}
```

Questa funzione crea però un problema, essa crea 2 oggetti locali, `resto` e `selezionate`, che a fine funzione verranno deallocati:

- La funzione memorizza su "resto" la lista delle telefonate non selezionate e le assegna a `b` tramite una copia profonda;
⇒ Quando la funzione termina, la memoria dinamica allocata per la lista associata a "resto" non viene recuperata.
- Inoltre la funzione memorizza nell'oggetto "selezionate" la lista delle telefonate selezionate.
⇒ Facendo `return selezionate`; viene richiamato il costruttore di copia che costruisce un oggetto anonimo che verrà distrutto non appena verrà usato come valore di ritorno.
Dato che anche il costruttore di copia effettua una copia profonda, non viene recuperata la memoria dinamica delle due liste di "selezionate" e dell'oggetto anonimo.

Perciò ci saranno 3 liste allocate dinamicamente che non vengono deallocate!

3.5 Tempo di vita delle variabili

E' l'intervallo di tempo in cui la variabile viene mantenuta in memoria durante l'esecuzione del programma. Possono esserci:

- Variabili di classe automatica:** definite all'interno di qualche blocco (variabili locali dentro una funzione e i parametri formali passati ad esse) che vengono allocate nello stack quando l'esecuzione raggiunge la loro definizione e vengono deallocate quando termina l'esecuzione del blocco in cui sono definite.
- Variabili di classe statica:** comprendono i campi dati statici, le variabili statiche definite in qualche blocco e le variabili globali (definite all'esterno di ogni funzione e classe). Variabili globali e campi dati statici vengono allocate all'inizio dell'esecuzione del programma, mentre le variabili statiche interne ad un blocco sono allocate quando l'esecuzione raggiunge la loro definizione.
Tutte le variabili di questo tipo vengono deallocate a fine esecuzione del programma.
- Variabili dinamiche:** sono allocate nello heap quando viene eseguito l'operatore `new` e deallocate con `delete`.

Le variabili con lo stesso tempo di vita, cioè quelle definite all'interno dello stesso blocco, vengono deallocate nell'ordine inverso a quello di allocazione.

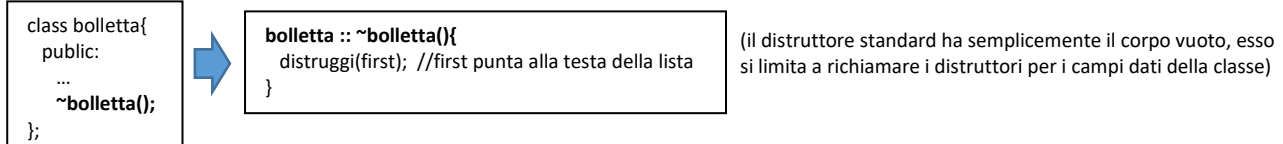
3.6 Distruttore

Per le variabili che sono oggetti di qualche classe ⇒ -**allocazione** significa **invocazione di un costruttore**;
-**deallocazione** significa **invocazione del distruttore**;

Quando termina il tempo di vita di un oggetto x, viene perciò richiamato automaticamente il **distruttore standard** che rilascia la memoria occupata da x. Bisogna però prestare molta attenzione, perché se l'oggetto x include dei campi puntatore allora il distruttore standard si limita a rilasciare la memoria occupata dai campi puntatore e non dealloca la memoria a cui puntano, perciò molte volte non è corretto.
(Il distruttore standard vale anche per i tipi primitivi o derivati)

E' possibile ridefinire il distruttore in modo che esso effettui una **distruzione profonda** degli oggetti, deallocando anche la memoria a cui puntano i campi puntatore, in modo che se un oggetto x include un campo dati p che punta alla testa di una lista, con la distruzione profonda di x si deallocherà tutta la lista puntata da p.

Il distruttore è un metodo senza parametri e senza tipo di ritorno inserito nella parte pubblica, è identificato da nome classe e simbolo ~.



I distruttori vengono invocati:

- Per gli oggetti di classe statica ⇒ al termine del programma(a fine main);
- Per gli oggetti di classe automatica definiti in un blocco ⇒ al termine del blocco in cui sono definiti;
- Per gli oggetti dinamici ⇒ quando viene eseguito l'operatore delete, altrimenti al termine del programma;
- Per gli oggetti che sono campi dati di qualche oggetto x ⇒ quando x viene distrutto;
- Per gli oggetti con lo stesso tempo di vita ⇒ sono distrutti nell'ordine inverso a quello di creazione.

Vengono invocati:

- Sulle variabili locali di una funzione al termine dell'esecuzione della stessa funzione;
- Sui parametri di una funzione passati per valore al termine dell'esecuzione della funzione;
- Sull'oggetto anonimo ritornato come risultato da una funzione non appena esso sia stato usato.

In particolare, la distruzione al ritorno di una chiamata di funzione F() avviene:

1. vengono distrutte le variabili locali a F();
2. viene distrutto l'oggetto anonimo ritornato per valore da F() subito dopo essere stato usato;
3. vengono distrutti i parametri di F() passati di valore.

Data una funzione F (T1 f1, T2 f2,..., Tn fn) con più di un parametro formale ⇒ ad ogni invocazione di F(a1, .. , an) l'ordine seguito per passare i parametri attuali a1,...,an avviene da destra verso sinistra, perciò la lista di inizializzazione per i parametri va dall'ultimo al primo e dato che al momento della distruzione dei parametri passati per valore è inverso rispetto alla loro inizializzazione, l'ordine di distruzione dei parametri passati per valore è da sinistra verso destra.

Data la classe C con campi dati x1,...,xn, se viene distrutto un oggetto di tipo C, viene automaticamente invocato il distruttore di C.

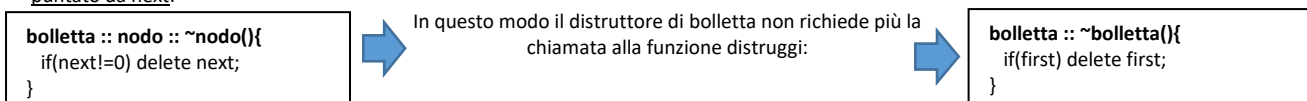
Il suo comportamento è:

1. viene eseguito il corpo del distruttore di C, se questo esiste;
2. vengono richiamati i distruttori per i campi dati nell'ordine inverso alla loro lista di dichiarazione. Per i tipi primitivi viene semplicemente rilasciata la memoria, per i tipi classe viene invocato il rispettivo distruttore standard o ridefinito.

⇒ Aggiungendo sulla classe il distruttore ridefinito ~bolletta(); ora al termine della funzione Chiamate_A(); i due oggetti locali "selezionate" e "resto" vengono automaticamente distrutti dato che il distruttore ridefinito invoca il metodo statico privato distruggi che a sua volta esegue il delete su tutti gli elementi della lista. Quello che succede in dettaglio è:

1. delete p; provoca l'invocazione del distruttore della classe nodo e dato che questo non è stato ridefinito ⇒ viene richiamato il distruttore di default di nodo;
2. il distruttore di default di nodo richiama a sua volta per ogni campo dati di nodo, cioè info e next, i rispettivi distruttori.
⇒ Il campo next viene deallocato, mentre per info ⇒ viene invocato il distruttore della classe telefonata e siccome anch'esso è stato ridefinito ⇒ viene richiamato il distruttore standard di telefonata;
3. il distruttore standard di telefonata dealloca l'intero numero e richiama il distruttore di orario per i campi inizio e fine, che sarà quindi il distruttore standard di orario;
4. il distruttore standard della classe di orario dealloca il campo dati intero "sec".

Poiché telefonata e orario non hanno campi dati puntatore, mentre nodo ha il campo next distrutto con la chiamata distruggi(p->next);, un modo più elegante di procedere sarebbe stato quello di ridefinire anche il distruttore per la classe nodo in modo che provveda a distruggere l'oggetto puntato da next:

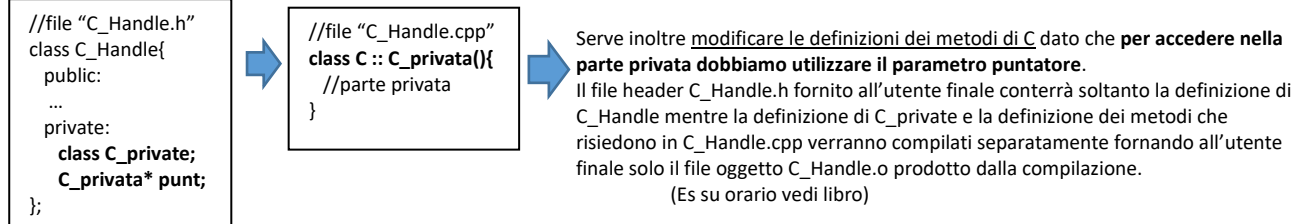


NB: Con questa modifica, il comportamento di Estrai_Una(); cambia dato che delete p; causa la distruzione profonda della lista di nodi puntata dal campo dati first dell'oggetto di invocazione. Serve una modifica:

```
telefonata :: bolletta :: Estrai_Una(){
    nodo* p= first;
    first= first->next;
    telefonata aux=p->info;
    p->next=0; //isolo primo nodo
    delete p;
    return aux;
}
```

3.7 Nascondere la parte privata di una classe (NON SI USA MOLTO)

Si fa dichiarando una classe C_privata all'interno della parte privata della classe C, dichiarando nella parte privata di C un puntatore a tale classe.



NB: questa tecnica può causare problemi di interferenza, visto che le classi hanno un puntatore come campo dati.

3.8 Array di oggetti

Si possono definire array statici o dinamici di oggetti, quando vengono costruiti viene richiamato il corrispondente costruttore di default per ogni oggetto dell'array mentre per la distruzione viene richiamato implicitamente il distruttore per ogni oggetto dell'array.

Un array può essere costruito a partire da una lista di inizializzazione in modo analogo a quelli di P1, se la lista non è completa allora per gli elementi dell'array per cui non viene fornito un oggetto per la costruzione viene richiamato il relativo costruttore.

3.9 Cast

Per questioni di compatibilità con il C++, per le conversioni esplicite la notazione () è stata ereditata dal C.

Es:

```
float x;  
char c;  
void *p;  
float y = (float) c;  
nodo* q = (nodo*) p;
```

Le conversioni di tipo sono però operazioni "pericolose" in quanto potrebbero portare a perdite di informazione.

Il C++ standard ha perciò introdotto una notazione esplicita più visibile e differenziata tra le varie tipologie di cast.

La sintassi è :

`nome_cast <Tipo> (expr)`

dove: -nome: indica la tipologia del cast che può essere: const, static, reinterpret o dynamic;
-Tipo: indica il "tipo target" in cui deve essere convertito il valore dell'espressione expr;
-expr: indica l'espressione da convertire.

3.9.1 Static Cast

`static_cast <Tipo> (expr)`

Permette di rendere esplicito l'uso di tutte le conversioni, implicite o esplicite, previste o permesse dal linguaggio o definite dal programmatore.

-Tutte le conversioni oggetto dello static cast si basano su info di tipo statico, cioè disponibile a compile-time.

-Rende visibile l'uso della conversione in situazioni potenzialmente pericolose, come per le conversioni tra tipi predefiniti con perdita di informazioni (dette "narrowing conversions", ad es da long ad int) che di solito provocano soltanto un warning. Usando lo static_cast tali warning saranno evitati.

-Permette il cast tra puntatori e tipi qualsiasi (molto pericoloso) e in particolare dal tipo puntatore generico void* a Tipo*.

Le conversioni implicite "safe" (cioè senza perdita di info) sono:

Es:

```
double d = 3.14;  
int x = static_cast<int>(d); //narrowing conversion, ora x = 3.  
char c = 'a';  
int x = static_cast<int>(c); //castless conversion  
void* p;  
p = &d;  
double* q = static_cast<double*>(p); //conversione void* ⇒ T*  
int* r = static_cast<int*>(q); //castless conversion
```

```
T& ⇒ T  
T[] ⇒ T*  
T ⇒ const T  
T* ⇒ const T*  
float ⇒ double ⇒ long double  
char ⇒ short ⇒ int ⇒ long  
unsigned char ⇒ unsigned short ⇒ unsigned int ⇒ unsigned long
```

3.9.2 Const Cast

`const_cast <T*> (puntatore costante)` oppure `const_cast <T&> (riferimento costante)`

Permette di convertire un puntatore/riferimento ad un tipo const T in un puntatore/riferimento a T (rimuove quindi l'attributo const).

-Se si tenta di usare altri operatori di conversione diversi da questo, il C++ segnerà errore.

Es:

```
const int i = 5;  
int* p = const_cast<int*>(&i);  
  
void F(const C& x){  
    const_cast<C&>(x).metodo_non_costante();  
}  
  
int j = 7;  
const int* q = &j;
```

3.9.3 Reinterpret Cast

`reinterpret_cast <T*> (puntatore costante)` oppure `reinterpret_cast <T&> (riferimento costante)`

Si limita a reinterpretare a basso livello la sequenza di bit con cui è rappresentato il valore puntato da puntatore come fosse un valore di tipo T.

-Permette ogni tipologia di conversione tra puntatori e riferimenti.

-E' particolarmente pericoloso, permette situazioni estreme e con poco senso, ad es:

```
Classe c;  
int* p = reinterpret_cast<int*>(&c);  
const char* a = reinterpret_cast<const char*>(&c);  
string s(a);  
cout << s;
```

3.9.4 Dynamic Cast

`dynamic_cast <T*> (puntatore costante)` oppure `dynamic_cast <T&> (riferimento costante)`

Il "tipo dinamico" di puntatore o riferimento non è noto a tempo di compilazione ma soltanto a tempo di esecuzione. Si tratta quindi di una conversione che avviene dinamicamente a tempo di esecuzione. (Vedi CAP 6)

3.10 Funzioni Amiche

Se vogliamo definire l'operatore di output `operator<<` per la classe `bolletta`, si può fare:

```
ostream& operator<<(ostream& os, bolletta b){ //NB: b passato per valore
    os<<"Telefonate in bolletta: "<<endl;
    int i=1;
    while(!b.Vuota()){
        os<<i<<" "<<b.Estrai_Una()<<endl;
        i++;
    }
    return os;
}
```

Questa soluzione è corretta in quanto utilizza soltanto metodi pubblici della classe `bolletta` e l'operatore `operator<<` della classe `telefonata`.

Questa soluzione è però inefficiente in quanto effettuare una copia profonda di una lista solo per percorrerla e stampare gli elementi è uno spreco. Il problema è che una funzione esterna non può accedere ai campi privati di una classe e quindi non può percorrere la lista dato che "first" di `bolletta` è un campo privato.

La funzione `operator<<` benché sia funzione esterna, è da considerarsi a tutti gli effetti un membro della classe. E' quindi possibile dichiarare una funzione esterna alla classe detta **funzione amica** tramite **friend** che **permette di accedere alla parte privata della classe**. Es:

```
//file "bolletta.h"
class bolletta{
    ..
    friend ostream& operator<<(ostream&, const bolletta&);
    ...
}
```



```
//file "bolletta.cpp"
ostream& operator<<(ostream& os, const bolletta& b){
    os<<"Telefonate in bolletta : "<<endl;
    bolletta :: nodo* p=b.first; //uso amicizia
    int i=1;
    while(p){
        os<< i++ <<" " <<p->info<<endl;
        p=p->next;
    }
    return os;
}
```

3.11 Classi amiche e iteratori

Nella manipolazione degli oggetti di una collezione è necessario poter accedere agli elementi della collezione in modo da poterli scorrere. Si può perciò definire una nuova **classe iteratore** i cui oggetti sono "indici" di elementi della classe contenitore (stessa idea degli indici interi che scorrono array).

-Se la classe C è amica di una classe D \Rightarrow allora a tutti i membri di C è permesso l'accesso alla parte privata di D.

Es: Vogliamo definire una classe iteratore i cui oggetti rappresentano gli indici ai nodi degli oggetti della classe contenitore.

```
class contenitore{
private:
    class nodo{
    public:
        int info;
        nodo* next;
        nodo(int x, nodo* p): info(x), next(p) { }
    };
    nodo* first;

public:
    contenitore(): first(0) { }
    void aggiungi_in_testa(int x) { first= new nodo(x,first); }
};
```



```
class iteratore{
private:
    contenitore::nodo* punt; //per amicizia
public:
    bool operator==(iteratore i) const{
        return punt==i.punt;
    }
    iteratore& operator++(){ //operatore ++ prefisso
        if(punt) punt= punt->next;
        return *this;
    }
};
```

In questo caso però, sia il campo `punt` che i metodi di `iteratore` accedono alla parte privata di `contenitore`
 \Rightarrow Bisogna perciò **definire la classe iteratore come classe pubblica interna alla classe contenitore**.

- La classe contenitore metterà a disposizione dei metodi pubblici per definire gli iteratori “iniziali” e “finali” su un certo oggetto e ridefinirà l’operatore di indicizzazione operator[] con un parametro it di tipo iteratore che permette di accedere all’info memorizzata all’indice it.
 - Dato che questi metodi dovranno accedere al puntatore privato punt di iteratore
 - ⇒ Bisogna garantire l’accesso dichiarando la classe contenitore amica di iteratore.

```
class contenitore{
    friend class iteratore; //dichiarazione amicizia
private:
    class nodo{
    public:
        int info;
        nodo* next;
        nodo(int x, nodo* p): info(x), next(p) { }
    };
    nodo* first;

public:
    contenitore();

    class iteratore{
        friend class contenitore; ⇒ dichiarazione di amicizia
    private:
        contenitore::nodo* punt; ⇒ per amicizia
    public:
        bool operator==(iteratore i) const{
            return punt==i.punt;
        }
        iteratore& operator++(){
            if(punt) punt= punt->next;
            return *this;
        }
        iteratore begin() const; ■ NB: la dichiarazione di iteratore deve sempre precedere le
        iteratore end() const;             dichiarazioni dei metodi che usano tale tipo
        int& operator[] (iteratore) const;

    };

    contenitore::iteratore contenitore::begin() const{
        iteratore aux;
        aux.punt= first; //per amicizia
        return aux;
    }

    contenitore::iteratore contenitore::end() const{
        iteratore aux;
        aux.punt=0;
        return aux;
    }

    int& contenitore::operator[](contenitore::iteratore it){
        return it.punt->info; //per amicizia
    }

    void aggiungi_nodo(int x);
};
```

Ora si possono utilizzare gli iteratori della classe contenitore su funzioni esterne:

```
int somma_elementi(const contenitore& c){
    int s = 0;
    for(contenitore::iteratore it = c.begin(); it != c.end(); ++it)
        s += c[it];
    return s;
}
```

Es: Applichiamo l'idea di iteratore su bolletta:

```
//file "bolletta.h"
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include "telefonata.h"

class bolletta{
    friend class iteratore;

private:
    class nodo{
    public:
        nodo();
        nodo( const telefonata& x, nodo* p): info(x), next(p) {}
        telefonata info;
        nodo* next;
        ~nodo();
    };
    nodo* first;
    static nodo* copia(nodo*);
    static void distruggi(nodo*);
public:
    class iteratore{
        friend class bolletta;
    private:
        bolletta::nodo* punt; //nodo puntato dall'iteratore
    public:
        bool operator==(iteratore) const; //metodi obbligatori sempre presenti
        bool operator!=(iteratore) const;
        iteratore& operator++( ); //operatore ++ prefisso
        iteratore& operator++(int); //operatore ++ postfisso, int fittizio aggiunto per
        distinguerli
    };
    bolletta(): first(0) {}
    ~bolletta(); //distruttore profondo
    bolletta(const bolletta&); //copia profonda
    bolletta& operator=(const bolletta&); //assegnazione profonda
    bool Vuota() const;
    void Aggiungi_Telefonata(telefonata t);
    void Togli_Telefonata(telefonata t);
    telefonata Estrai_Una( );
    iteratore begin( ) const; //metodi per usare iteratore
    iteratore end( ) const;
    telefonata& operator[ ](iteratore) const;
};
#endif
```



```
//file "bolletta.cpp" solo i metodi mancanti
#include "bolletta.h"

bool bolletta::iteratore::operator==(iteratore i) const{
    return punt==i.punt;
}

bool bolletta::iteratore::operator!=(iteratore i) const{
    return punt!=i.punt;
}

bolletta::iteratore& bolletta::iteratore::operator++(){ //prefisso
    if(punt) punt=punt->next; //side-effect
    return *this;
} //se punt==0 non fa nulla

bolletta::iteratore bolletta::iteratore::operator++(int){
    iteratore aux=*this;
    if(punt) punt=punt->next; //side-effect
    return aux;
}

bolletta::iteratore bolletta::begin() const{
    iteratore aux;
    aux.punt=first; //per amicizia
    return aux;
}

bolletta::iteratore bolletta::end() const{
    iteratore aux;
    aux.punt=0; //per amicizia
    return aux;
}

telefonata& bolletta::operator[ ] (bolletta::iteratore it){
    return (it.punt->info; //per amicizia
} //NB:nessun controllo su i.punt!=0
```

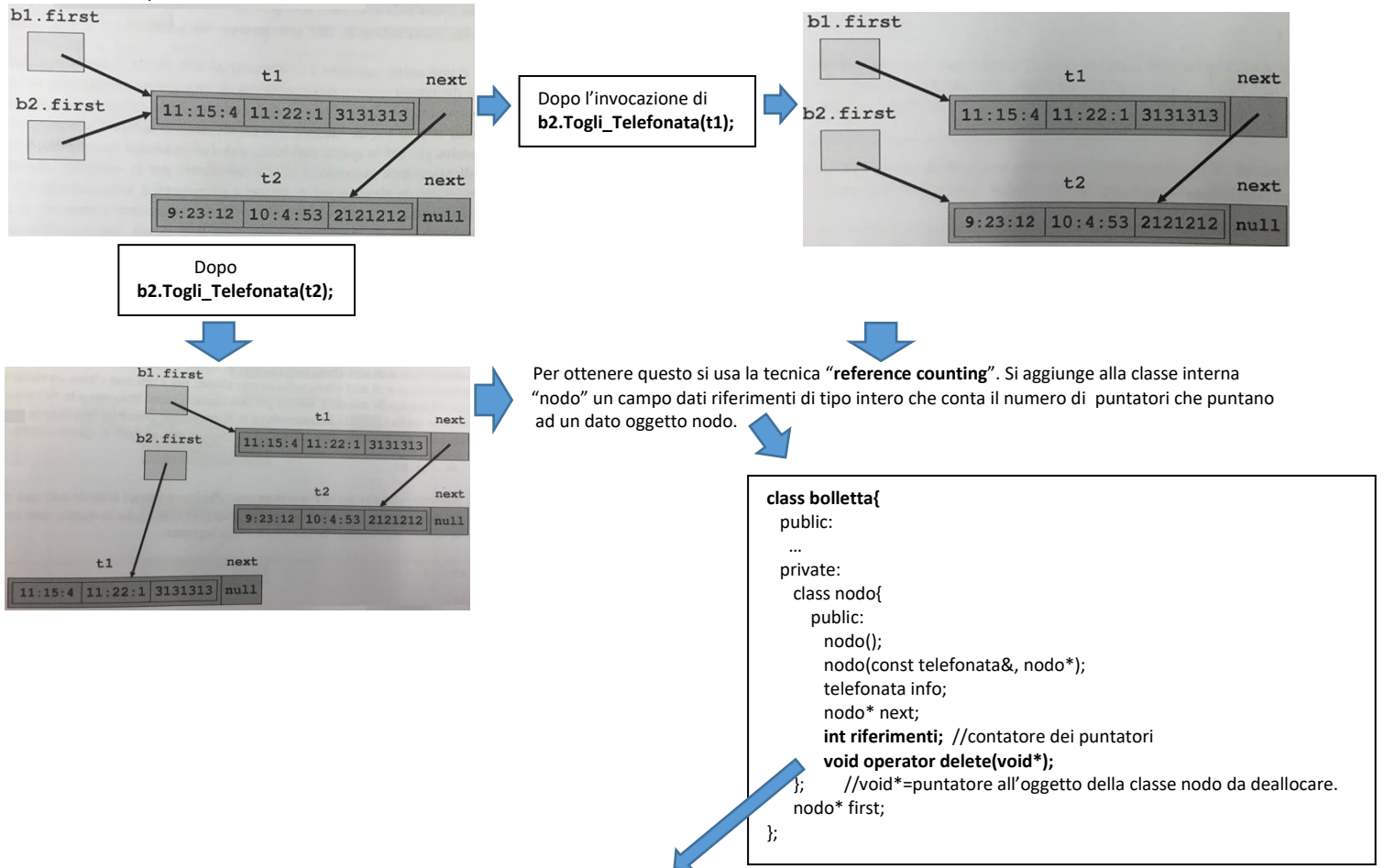
Es: funzione che calcola la somma delle telefonate di una bolletta tramite l'uso di iteratore.

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


3.13 Condivisione Controllata della memoria

La ridefinizione dei costruttori di copia e di assegnazione per bolletta evitano completamente condivisione e interferenze di memoria. Tuttavia, eseguire copie profonde seguite da distruzioni profonde può essere molto costoso. Ad esempio la copia profonda risulta inutile quando passiamo un parametro per valore ad una funzione che non lo modifica.

Data questa condivisione di memoria vorremmo avere:



NB: Bisogna inoltre ridefinire l'operatore delete della classe nodo in modo tale che esso si limiti a decrementare di una unità il campo riferimenti, e soltanto quando questo è 0 rilasci definitivamente la memoria:

```
void bolletta::nodo::operator delete(void* p){
    if(p){ //se c'è qualcosa da deallocare
        nodo* q=static_cast<nodo*>(p); //cast esplicito
        q->riferimenti--; //sappiamo che q->riferimenti è sicuramente maggiore di 0
        if(q->riferimenti==0){ //allora dealloco definitivamente il nodo
            delete q->next; //invoco ricorsivamente la delete ridefinita su q->next poiché sto per eliminare un puntatore al nodo puntato da q->next
            ::delete q; //uso delete standard per deallocare il nodo
        }
    }
}
```

NB: l'overloading dell'operatore delete per una classe C non ha effetti sull'operatore delete di altre classi. Per invocare la delete standard per C basta usare l'operatore di scope globale: `::delete p;`

Implementare correttamente la memoria condivisa è molto importante dato che potrebbero esserci gravi inconsistenze nel caso non si mantenga aggiornato il campo riferimenti.

Riepilogo Bolletta con aggiunta dei puntatori con il count:

//file "bolletta.h" con la dichiarazione di bolletta

```
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include "telefonata.h"
```

```
class bolletta{
private:
    class nodo{
    public:
        telefonata info;
        nodo* next;
        int riferimenti;
        nodo();
        nodo( const telefonata& , nodo* ); //inserisce telefonata in testa a lista
        void operator delete(void*);
    };
    nodo* first;
public:
    bolletta(): first(0) { }
    bolletta(const bolletta&); //costruttore di copia
    ~bolletta(); //distruttore profondo
    bool Vuota() const;
    void Aggiungi_Telefonata(const telefonata& );
    void Togli_Telefonata(const telefonata& );
    telefonata Estrai_Una();
    bolletta& operator=(const bolletta&); //assegnazione profonda
};
#endif
```

NB: rispetto a bolletta originale è stata ridefinita la classe nodo ed il puntatore first alla testa della lista. Inoltre sono stati eliminati i metodi statici privati copia e distruggi che servivano per effettuare copie e distruzioni profonde.

//la modifica di Togli_Telefonata() è delicata. L'idea è quella di fare una copia profonda parziale della lista fino al nodo contenente la telefonata da rimuovere.

Problema: La copia risulterebbe inutile nel caso non si trovasse una telefonata da togliere.

void bolletta::Togli_Telefonata(const telefonata & t){

```
nodo* p=first, *prec=0, *q=0;
first=0;
while(p&&! (p->info==t)){ //copia degli elementi precedenti a quello da togliere
    if(q) delete q;
    q=new nodo(p->info, p->next);
    q->riferimenti++;
    if(prec == 0){ first=q;
        first->riferimenti++;
    }else{ if(prec->next) delete prec->next;
        prec->next=q;
        prec->next->riferimenti++;
    }
    if(prec) delete prec; //ora aggiorno prec e p per il ciclo while
    prec=q;
    prec->riferimenti++;
    if(p) delete p;
    p=q->next;
    if(p) p->riferimenti++;
} //fine while
//se p==0 la telefonata non è stata trovata
if(p) //allora rimuovo il nodo puntato da p
if(prec==0){ //allora t era in testa
    first=p->next;
    if(first) first->riferimenti++;
}else{ if(prec->next) delete prec->next;
    prec->next=p->next;
    if(prec->next) prec->next->riferimenti++;
}
if(p) delete p; //elimino puntatori locali
if(prec) delete prec;
if(q) delete q;
}
```

bolletta& bolletta::operator=(const bolletta& b){

```
if(this != &b){
    if(first) delete first;
    first=b.first;
    if(first) first->riferimenti++;
}
return *this;
}
```

//file "bolletta.cpp"

```
#include <iostream>
#include "bolletta.h"
```

bolletta::nodo::nodo(): next(0), riferimenti(0) { }
//per il campo info interviene il costruttore standard

bolletta::nodo::nodo(const telefonata& t, nodo* p): info(t), next(p), riferimenti(0){
if(next) next->riferimenti++; //boh se non funziona riguarda libro pg98
}

void bolletta::nodo::operator delete(void* p){
if(p){ //se c'è qualcosa da deallocare
 nodo* q=static_cast<nodo*>(p); //cast esplicito
 q->riferimenti--; //sappiamo che q->riferimenti è sicuramente maggiore di 0
 if(q->riferimenti==0){ //allora dealloco definitivamente il nodo
 delete q->next; //chiamata ricorsiva della delete ridefinita
 ::delete q; //uso delete standard per deallocare il nodo
 }
}

bolletta::bolletta(const bolletta& b): first(b.first){
if(first) first->riferimenti++;
} //il costruttore di copia di bolletta ora provoca condivisione di memoria e non più
copia profonda, quindi si limita a copiare il puntatore first e incrementa
l'oggetto puntato

bolletta::~bolletta(){ //distruttore
if(first) delete first; //richiama versione ridefinita di delete
}

bool bolletta::Vuota() const{ return first==0; }

void bolletta::Aggiungi_Telefonata(const telefonata& t){

```
if(first) first->riferimenti--;  
//NB: in questo punto c'è una interferenza temporanea, first->riferimenti--  
è necessario perché la new seguente invoca il costruttore nodo e  
quindi comporta l'incremento del campo first->riferimenti.  
first=new nodo(t,first);  
first->riferimenti++; //elimino l'inconsistenza  
}
```

telefonata bolletta::Estrai_Una(){ //PRE=(bolletta di invocazione non vuota, first!=0)

```
telefonata aux=first->info;  
nodo* p=first;  
//p->riferimenti++;  
//first->riferimenti--;  
first=first->next;  
if(first->next) first->riferimenti++;  
delete p;  
return aux;  
}
```

3.13.1 Puntatori Smart

La difficoltà principale nella tecnica del reference counting è quella di mantenere aggiornato il campo riferimenti degli oggetti di tipo nodo. Quando assegniamo ad un puntatore p di tipo nodo* l'indirizzo di un x di nodo, non sempre è possibile inserire l'istruzione che aggiorna x.riferimenti nel punto esatto in cui avviene l'assegnazione o la distruzione del puntatore. Ciò causa inconsistenze momentanee.

Per rendere automatica la gestione del campo riferimenti, basterebbe ridefinire assegnazione, costruttore di copia e distruttori del parametro nodo. Ciò non è possibile in quanto i puntatori sono tipi derivati e non oggetti di una classe definita dal programmatore.

La soluzione è perciò quella di definire una **classe da usare come puntatore smart** che punta alla classe nodo che contiene un unico campo dati di tipo nodo* in cui ridefiniamo assegnazione, costruttore di copia e distruttore che aggiornano il campo riferimenti di nodo.

```
//file "bolletta.h" con la dichiarazione di bolletta
```

```
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include <iostream>
#include "telefonata.h"
using std::ostream;
```

```
class bolletta{
```

```
public:
```

```
    bool Vuota() const;
    void Aggiungi_Telefonata(const telefonata&);
    void Togli_Telefonata(const telefonata&);
    telefonata Estrai_Una();
    friend ostream& operator<<(ostream&, const bolletta&);
```

```
private: //contiene la definizione della classe nodo e della classe smartp "puntatore furbo" a nodo
```

```
    class nodo; //dichiarazione incompleta di classe
```

```
    class smartp{
```

```
    public:
```

```
        nodo* punt; //unico campo dati di smartp
        smartp(nodo* p=0); //NB:Costruttore che: -Agisce da convertitore implicito da nodo* a smartp;
                                -Funziona anche da costruttore senza argomenti;

        smartp(const smartp&); //Costruttore di copia
        ~smartp(); //Distruttore
        smartp& operator=(const smartp&); //Assegnazione
        nodo& operator*() const; //Dereferenziazione
        nodo* operator->() const; //Accesso a membro
        bool operator==(const smartp&) const; //Uguaglianza
        bool operator!=(const smartp&) const; //Disuguaglianza
    }; //fine smartp
```

```
class nodo{ //dichiarazione completa di nodo
```

```
public:
```

```
    nodo();
    nodo(const telefonata&, const smartp&);
    telefonata info;
    smartp next; //smart pointer al nodo successivo
    int riferimenti;
};
```

```
    smartp first; //unico campo dati di bolletta
```

```
};
#endif
```

```
//file "bolletta.cpp"
```

```
#include "bolletta.h"
```

```
using std::ostream;
```

```
using std::endl;
```

```
//Definisco metodi di smartp
```

```
bolletta::smartp::smartp(nodo* p): punt(p){ if(punt) punt->referimenti++; } //COSTRUTTORE
```

```
bolletta::smartp::smartp(const smartp& s): punt(s.punt){ if(punt) punt->referimenti++; } //COSTRUTTORE DI COPIA
```

```
bolletta::smartp::~smartp(){ //DISTRUTTORE
```

```
    if(punt){
        punt->referimenti--;
        if(punt->referimenti==0) delete punt; //NB: la delete standard di nodo richiama il distruttore di smartp
    }
}
```

```
bolletta::smartp& bolletta::smartp::operator=(const smartp& s){ //ASSEGNAZIONE
```

```
    if(this!=&s){
        nodo* t=punt;
        punt=s.punt;
        if(punt) punt->referimenti++;
        if(t){
            t->referimenti--;
            if(t->referimenti == 0) delete t; //delete standard di nodo che richiama il distruttore di smartp
        }
    }
    return *this;
}
```

```
//Ridefinizione degli operatori
```

```
bolletta::nodo&& bolletta::smartp::operator*() const{ //OPERATORE DI DEREFERENZIAZIONE
    return *punt; //trasforma un oggetto smartp s nell'L-valore del nodo puntato da s.punt
}
```

```
bolletta::nodo* bolletta::smartp::operator->() const{ //OPERATORE SELEZIONE DI MEMBRO
    return punt;
}
```

```
bool bolletta::smartp::operator==(const smartp& p) const{ //OPERATORE UGUAGLIANZA
    return punt==p.punt;
}
```

```
bool bolletta::smartp::operator!=(const smartp& p) const{ //OPERATORE DISUGUAGLIANZA
    return punt!=p.punt;
}
```

```
//definizione dei due metodi costruttori di nodo
```

```
bolletta::nodo::nodo(): riferimenti(0){ } //invocazione di info() e next() implicite
```

```
bolletta::nodo::nodo(const telefonata& t, const smartp&p): info(t), next(p), riferimenti(0) { }
//next(p) invoca il costruttore di copia ridefinito di smartp
```

```
//definizione metodi di bolletta con i puntatori smart che gestiscono automaticamente il campo riferimenti.
```

```
bool bolletta::Vuota() const{ return first==0; } //cast implicito da 0 a smartp
```

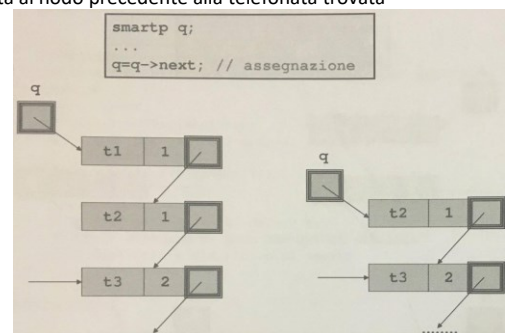
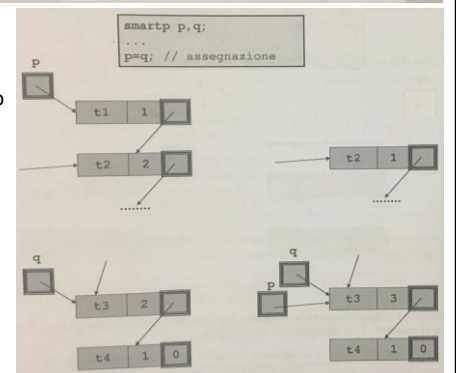
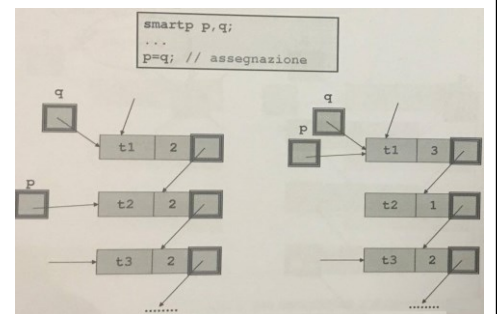
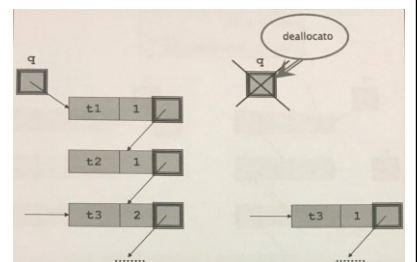
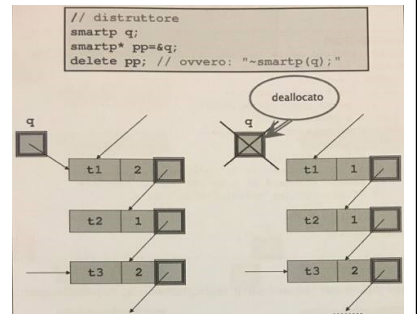
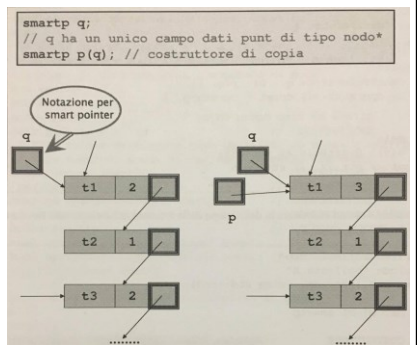
```
void bolletta::Aggiungi_Telefonata(const telefonata& t){ first=new nodo(t,first); } //cast implicito da nodo* a smartp
```

```
void bolletta::Estrai_Una(){ //PRE=( bolletta non vuota)
    telefonata aux=first->info;
    first=first->next; //NB: non bisogna fare la delete sul primo nodo
    return aux;
}
```

```
void bolletta::Togli_Telefonata(const telefonata& t){
    smartp p=first, prec, q; //p scorre la lista, prec è il nodo precedente al puntato da p, q punta alla copia del nodo
    smartp original=first; // first originale
    first=0;
    while(p!=0 && !(p->info==t)){
        q=new nodo(p->info, p->next); //faccio copia dei nodi che precedono quello da togliere
        if(prec==0) first=q; //e li inserisco nella nuova lista puntata da first
        else prec->next=q;
        prec=q; //aggiorno prec e p per il ciclo while
        p=p->next;
    }
    if(p==0) first=original; //se p==0 allora la telefonata non è stata trovata, ho copiato inutilmente la lista, perciò ora ripristino la situazione originale
    else if(prec==0) first=p->next; //se la telefonata t è stata trovata ed era in testa
    else prec->next=p->next; //altrimenti stacco la telefonata trovata e riattacco il resto della lista al nodo precedente alla telefonata trovata
} //NB: a fine funzione le variabili locali p,q e prec vengono deallocate implicitamente
```

```
ostream& operator<<(ostream& os, const bolletta& b){
```

```
    if(b.Vuota()) os<<"Bolletta Vuota"<<endl;
    else{ os<<"Telefonate in Bolletta:"<<endl;
        bolletta::smartp p=b.first; //per amicizia
        int i=1;
        while(p!=0){ os<<i++<<" "<<p->info<<endl;
            p=p->next;
        }
    } return os;
}
```



CAPITOLO 4 – Template

Il C++ è un linguaggio strongly typed: il programmatore è tenuto a specificare il tipo di ogni elemento sintattico che durante l'esecuzione denota un valore e far sì che il linguaggio garantisca che tale valore sia utilizzato in modo coerente con il tipo specificato.

In certe occasioni, la tipizzazione forte può essere troppo vincolante, perciò, se usate in modo corretto, le conversioni implicite ed esplicite possono alleviare tale vincolo.

Ci sono varie situazioni che non possono essere risolte mediante conversioni di tipo, ad es se vogliamo definire una **funzione che calcoli il min** tra due valori dobbiamo necessariamente definire una **funzione diversa per ogni tipo di valori int, float, orario, string** etc.

Una soluzione attraente ma pericolosa può essere la definizione di macro istruzioni che vengono espanse nella fase di precompilazione usando:

```
#define min(a,b) ( (a) < (b) ? (a) : (b) )
```

min(10,20) con 10 < 20 ? 10 : 20
significa: 10 è minore di 20? Se sì ritorna 10, altrimenti 20
min(3.5,2.1); analogo

Questo funziona nei casi semplici, la soluzione migliore sta nell'usare i template di funzione.

4.1 Template di Funzione

Assieme ai template di classe, i template di funzione implementano in C++ l'idea del cosiddetto polimorfismo parametrico.

E' essenzialmente la descrizione di un metodo, che il compilatore può usare per generare automaticamente istanze particolari di una funzione che differiscono per il tipo degli argomenti.

Un template di funzione può essere sia una funzione globale che un metodo della classe.

La "costruzione" di una specifica funzione da un template per dei dati parametri attuali di tipo e valore viene detta **istanziamento del template**.

Si definisce:

```
template <class T>  
T min(Tipo a, Tipo b){  
    return a < b ? a : b;  
}
```

```
int main() {  
    int i, j, k;  
    orario r, s, t;  
}
```

■ **istanziamento implicito del template:**

```
k = min(i, j);  
t = min(r, s);
```

■ **istanziamento esplicito del template:**

```
k = min<int>(i, j);  
t = min<orario>(r, s);
```

La keyword "template" appare sia all'inizio delle dichiarazioni che nelle definizioni di un template funzione. Essa è seguita dalla lista dei parametri del template separati da virgole e racchiusi tra i simboli < >.

I parametri possono essere:

- parametri di tipo**: preceduti da "class" (oppure dal nome del tipo, ad es "int") e che dovranno essere istanziati con un tipo qualsiasi;
- parametri valore**: preceduti dal tipo di appartenenza e che devono essere istanziati con un valore costante del tipo indicato. Il nome del parametro valore usato può essere usato solo come valore costante interno alla definizione del template.

■ **Istanziamento implicito**: se i tipi e i valori effettivi dei parametri di tipo e valore del template sono dedotti automaticamente dai parametri attuali passati come argomenti al template.

Es:

```
int main() {  
    int i, j, k;  
    orario r, s, t;  
  
    k = min(i, j);  
    t = min(r, s);  
}
```

Inoltre, il tipo di ritorno dell'istanza del template non viene considerato nella deduzione degli argomenti:

```
int main() {  
    int i, j;  
    double d;  
  
    d = min(i, j);  
}
```

//istanzia: int min(int, int) e quindi usa la **conversione int ⇒ double**

Nella deduzione degli argomenti sono ammesse **4 tipi di conversioni** dal tipo dell'argomento attuale al tipo dei parametri del template:

1. Conversione **da L-valore a R-valore**, da T& ⇒ T;
2. Conversione **da array a puntatore**, cioè da T[] ⇒ T*;
3. Conversione di **qualificazione costante**, da T ⇒ const T;
4. Conversione **da R-valore a riferimento costante**, cioè da R-valore di tipo T ⇒ const T&.

NB: L'algoritmo di deduzione degli argomenti esamina tutti gli argomenti nella chiamata del template da sinistra verso destra, perciò:

- Se si trova uno stesso parametro T del template che appare più volte come parametro di tipo
⇒ allora l'argomento del template dedotto per T da ogni argomento attuale deve essere esattamente lo stesso.

Es:

```
int main() {  
    int i;  
    double d, e;  
    e = min(i, d);  
}
```

⇒ NO, deduce due diversi argomenti del template, int e double

■ **Istanziazione Esplicita:** se si specificano esplicitamente gli argomenti di un template.

NB: Nell'invocazione di un template di funzione F istanziato esplicitamente è **possibile applicare qualsiasi conversione esplicita di tipo** per convertire un parametro attuale per F nel tipo del corrispondente parametro dell'istanza implicita di F.

```
int main() {
    int i;    double d,e;
    e = min<double>(i, d); ⇒ OK, ora istanzia double min(double,double) e quindi converte implicitamente i da int a double
}
```

Di solito si usa l'istanziazione esplicita solo quando è necessario **risolvere ambiguità** o **per usare particolari istanze** in contesti in cui la deduzione degli argomenti non è possibile.

4.2 Modelli di Compilazione dei template funzione

La definizione di un template funzione è solo uno schema per la definizione di un numero potenzialmente infinito di funzioni.

■ Una definizione di template di funzione non può essere compilata in codice macchina che possa essere eseguito.

Il compilatore, quando incontra una def di template di funzione:

```
template <class T>
T min(T a, T b){ return a<b ? a : b; }
```

si limita a memorizzare una rappresentazione interna di tale definizione, senza effettuare alcuna vera compilazione in linguaggio macchina eseguibile.

Soltanto quando il compilatore incontrerà un uso effettivo del template funzione genererà codice macchina corrispondente alla particolare istanza di funzione utilizzata.

4.2.1 Compilazione per Inclusione

In questo caso le **definizioni dei template vengono messe in un file header**, che verranno inclusi in ogni file in cui vengono istanziati i template. Il compilatore quindi usa queste definizioni per generare codice di tutte le istanze del template usate nel file che sta compilando.

2 problemi:

- Il file header contiene tutti i dettagli della definizione del template che risultano quindi visibili all'utente. Ciò va contro il principio dell'information hiding ⇒ Non c'è soluzione.
- Se la stessa istanza di un template viene usata in più file compilati separatamente, il codice per tale istanza viene generato più volte dal compilatore. Ciò non crea problemi durante l'esecuzione dato che quando il linker collega i file compilati separatamente usa soltanto una delle istanze compilate, mentre le altre vengono ignorate.
 - inoltre la compilazione ripetuta delle stesse istanze e l'inclusione di file header troppo lunghi rallenta la compilazione.

Soluzione ⇒ dichiarazioni esplicite di istanziazione: alcuni compilatori permettono di selezionare una particolare opzione che impedisce la generazione automatica del codice delle istanze dei template. Bisogna quindi "forzare" il compilatore a generare istanze del template che vengono effettivamente usate nel programma:

```
template <class T>
T min(T a, T b){ return a<b ? a : b; }
```

template int min(int, int); ⇒ dichiarazione esplicita di istanziazione al tipo int

La definizione del template deve essere messa nel file in cui compare la dichiarazione esplicita di istanziazione, ciò forza il compilatore a generare il codice dell'istanza del template relativa al tipo int.

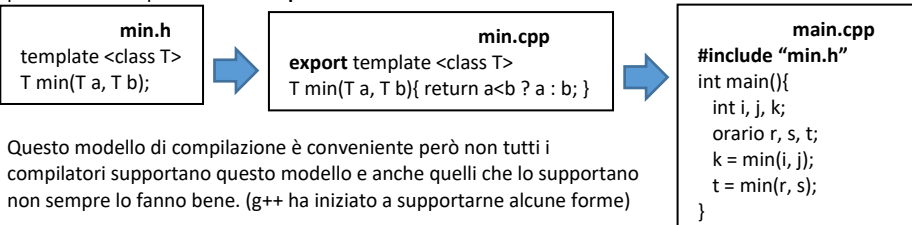
- Per gli altri file che usano la stessa istanza del template, viene fornito un parametro al compilatore e gli comunica che una dichiarazione esplicita di istanziazione compare in un altro file e che il template di funzione non deve essere istanziato. Nel caso di g++ si usa:

g++ -fno-implicit-templates

4.2.2 Compilazione per Separazione

In questo caso **nel file header vengono messe soltanto le dichiarazioni dei template**, mentre **le definizioni vengono messe in un file esterno** che può essere "compilato separatamente".

Per avvisare il compilatore che la definizione del template di funzione è necessaria per generare istanze del template usate in altri file, bisogna farla precedere dalla parola chiave **export**. Es:



4.3 Template di Classe

E' la **descrizione di un modello** che il compilatore può usare **per generare automaticamente istanze particolari di una classe che differiscono per il tipo di alcuni membri propri della classe.**

Es: vogliamo una classe "Queue" che implementa una coda con politica FIFO. Tale classe deve implementare una coda in cui gli elementi vengono estratti nello stesso ordine in cui sono inseriti.

```
class Queue{           // "Tipo" indica il tipo degli elementi della coda
public:
    Queue();
    ~Queue();
    bool is_empty( ) const; //controlla se la coda è vuota
    void add(const Tipo&); //aggiunge un item in coda
    Tipo remove( ); //rimuove item alla coda
private:
    ....
};
```

Il compilatore quando compila Queue ed i suoi metodi, deve conoscere a quale tipo effettivo ci si vuole riferire. Se vogliamo definire una coda di interi, dobbiamo sostituire "int" al posto di "Tipo" in tutti i punti della definizione nella classe in cui compare.

Se nello stesso programma dobbiamo usare sia code di interi che di stringhe, dobbiamo scrivere due definizioni distinte della classe con due nomi distinti, QueueInt e QueueString.

Duplicare le classi allunga inutilmente le righe del programma e rende difficile la modifica e il riutilizzo del software.

Bisogna quindi **usare i template, parametrizzando la classe rispetto ai tipi**, così da evitare il duplicamento delle funzioni:

```
template <class T>
class Queue{
public:
    Queue();
    ~Queue();
    bool is_empty( ) const; //controlla se la coda è vuota
    void add(const T&); //aggiunge un item in coda
    T remove( ); //rimuove item alla coda
private:
    ....
};
```

Per definire una coda qi di interi, qb di bollette e qs di stringhe:

```
Queue<int> qi;
Queue<bolletta> qb;
Queue<string> qs;
```

Come nei template di funzione, nei template di classe ci sono 2 tipologie di parametri:

- Parametri di tipo**: preceduti dalla keyword class(o typename);
- Parametri valore**: preceduti dal tipo del valore.

A differenza dei template di funzione, **con i template di classe è obbligatorio istanziare sempre in modo esplicito i parametri del template.**

I parametri di un template classe possono avere un **valore di default**. Es:

```
template <class T=int, int size=1024>
class Buffer{ ... };
```

```
int main( ){
    Buffer<> ib;           //Buffer<int, 1024>
    Buffer<string> sb;     //Buffer<string, 1024>
    Buffer<string,500> sbs; //Buffer<string, 500>
}
```

Es: Definizione completa della classe template Queue. Essa usa a sua volta un template di classe esterna QueueItem per rappresentare gli oggetti che sono elementi della coda. La coda verrà implementata come una lista puntata da due puntatori, uno al primo nodo e uno all'ultimo.

```
template <class T>
class QueueItem{ //template classe esterna di Queue
public:
    QueueItem(const T&);
    T info;
    QueueItem* next;
};
```

```
template<class T>
class Queue{
private:
    Queue<T>* primo; //due puntatori a primo e ultimo nodo
    Queue<T>* ultimo; //che vengono inizializzati al tipo di T
public:
    Queue();
    ~Queue();
    bool is_empty( ) const;
    void add(const T&);
    T remove( );
};
```


4.3.1 Istanziazione di un template di classe

La definizione di un template classe specifica come si possono costruire delle istanze di classe nel caso in cui vengano forniti i valori dei parametri del template.

■ Quando il compilatore incontra `Queue<int> qi;` ⇒ costruisce una classe “coda di interi” di nome `Queue<int>`, a meno che non lo abbia già fatto precedentemente. Dopo di che costruisce anche l’oggetto `qi` appartenente a tale classe.

■ Se poi incontra `Queue<string> qs;` ⇒ costruisce anche la classe “coda di stringhe” di nome `Queue<string>`.

Le due classi create, nonostante siano costruite mediante lo stesso template di classe, sono **due classi completamente distinte**.

Perciò `Queue<int>` non ha accesso alla parte privata di `Queue<string>` e viceversa.

NB: Nella dichiarazione o definizione di un template di classe (o di funzione), possono comparire sia nomi di istanze di template di classe, sia nomi di template di classe.

```
template<class T>
int F(Queue<T>& qT, Queue<string> qs); ⇒ Queue<T> è template di classe e Queue<string> è istanza di template di classe
```

NB: Al di fuori di definizioni o dichiarazioni di template, al contrario possono comparire solo nomi di istanze di template di classe.

La presenza del nome di una istanza di un template classe in un programma non è sufficiente al compilatore per generare tale istanza

⇒ Perciò è necessario che **il nome compaia in un contesto che richieda l’uso effettivo di tale definizione**.

Es:

```
template<class T> class Queue;

void Stampa(const Queue<int>& q){
    Queue<int>* pqi=&q;
    ...
}
```

Qui il compilatore non genera l’istanza `Queue<int>`, perché non è necessaria l’istanza della classe `Queue` per copiare un riferimento o un puntatore a `Queue`.

In questo caso basta solo la sua dichiarazione incompleta. (non serve la definizione)

Ciò vale per le definizioni di riferimenti e puntatori ad una classe, che appunto non richiedono un utilizzo effettivo della classe. (questo vale anche per una classe normale)

Un template di classe viene istanziato quando è presente la definizione del template che è necessaria per poter operare sugli oggetti della classe.

Es:

```
template<class T> class Queue{
    ...
};

void Stampa(const Queue<int> q){
    Queue<int> qi;
    ...
}
```

Qui il compilatore è “costretto” a generare l’istanza `Queue<int>` in quanto essa serve per allocare spazio per i due oggetti `q` e `qi`.

Deve quindi essere visibile la definizione della classe `Queue`.

Es:

```
template<class T> class Queue{
    ...
};

void Stampa(const Queue<int>& q){
    Queue<int>* pqi=&q;
    pqi++;
}
```

Anche in questo caso il compilatore è costretto a generare l’istanza `Queue<int>` dato che questa istanza serve per calcolare il `sizeof(Queue<int>)` di cui occorre incrementare il puntatore con `pqi++`.

4.3.2 Metodi di template di classe

Come per le classi normali, in un template di classe la definizione di un metodo può comparire sia all’interno (inline) che all’esterno della classe.

```
template<class T>
class Queue{
private:
    ...
public:    //Definizione inline
    Queue(): primo(0), ultimo(0){ }
    ...
};
```

oppure

```
template<class T>
class Queue{
private:
    ...
public:
    Queue();
    ...
};
```

```
//Definizione esterna
template <class T>
Queue<T>::Queue(): primo(0), ultimo(0){ }
```

Un metodo di un template di classe non viene istanziato quando viene istanziata la classe, ma viene istanziato solo quando il programma usa effettivamente quel metodo.

Es: definizione complete dei template di classe Queue e QueueItem

```
Queue.h
#ifndef QUEUE_H
#define QUEUE_H

template<class T>
class QueueItem{
public:
    QueueItem(const T& val): info(val), next(0){}
    T info;
    QueueItem* next;
};
```

```
Queue.h
template<class T>
class Queue{
public:
    Queue(): primo(0), ultimo(0){ }
    bool is_empty() const;
    void add(const T&);
    T remove;
    ~Queue();
private:
    QueueItem<T>* primo;
    QueueItem<T>* ultimo;
};

template <class T>                               Definizione metodi
bool Queue<T> :: is_empty() const{ return(primo==0); }

template<class T>
void Queue<T> :: add(const T& val){
    QueueItem<T>* p= new QueueItem<T>(val);
    if(is_empty()) primo = ultimo = p;
    else{ ultimo -> next = p; //aggiungo in coda
        ultimo=p;
    }
}

#include <iostream>
using std::cerr; using std::endl;

template<class T>
T Queue<T> :: remove(){
    if(is_empty()){ cerr << "impossibile fare remove() su coda vuota"<<endl;
        exit(1);
    }
    QueueItem<T>* p = primo;
    primo = primo -> next;
    T aux = p->info;
    delete p;
    return aux;
}

template<class T>
Queue<T> :: ~Queue(){
    while(!is_empty()) remove();
}
#endif
```

```
Queue.h
#include<iostream>
using std::count; using std::endl;
#include "Queue.h"

int main(){
    Queue<int>* pi = new Queue<int>;
    ⇒ dato che la new deve costruire un oggetto della classe vengono
    istanziati sia la classe Queue<int> ed il suo costruttore Queue<int>()
    int i;
    for(i=0; i<10; i++) pi -> add(i);
    ⇒ vengono istanziati i metodi add<int> e is_empty<int>, la classe
    QueueItem<int> e il suo costruttore QueueItem<int>()
    for(i=0; i<10; i++) cout<< pi -> remove()<<endl;
    ⇒ viene istanziato il metodo remove<int>
}
```

4.3.3 Dichiarazioni friend in template di classe

Ci sono tre tipologie di dichiarazioni friend per un template classe:

1. Classe o una Funzione non template dichiarate friend all'interno del template di classe C:

```
class A{ ... int fun(); .. }  
  
template<class T>  
class C{  
    friend int A::fun();  
    friend class B;  
    friend bool test();  
};
```

➡ (NB: obbligatorio dichiarare la classe A prima del template C per far sì che essa sia visibile a C, dato che include il metodo fun() dichiarato amico su C.)

➡ (Non serve definire B o test() prima del template C dato che non sono contenuti in una classe.)

➡ Il metodo A::fun(), la classe B, la funzione test() sono tutte istanze del template di classe C.

2. Template di classe o template di funzione friend associato (cioè ha tra i suoi parametri alcuni dei parametri del template C) dichiarati friend all'interno di un template di classe C, (caso più comune):

```
//dichiarazione dei template di classe associati a C:  
template<class T> class A{ ... int fun(); ... };  
template<class T> class B{ ... };
```

➡ (NB: i template classe A e B e la funzione fun devono essere visibili quando viene definita la classe template C)

```
//dichiarazione incompleta del template di classe C:  
template<class T1, class T2> class C;
```

➡ (NB: la classe C deve essere obbligatoriamente dichiarata prima della definizione di test)

```
//dichiarazione del template di funzione associato a C:  
template<class T2, class T2> bool test(C<T1, T2>);
```

```
template <class T1, class T2> //definizione di C  
class C{  
    friend int A<T1>::fun();  
    friend class B<T2>;  
    friend bool test<T1,T2>(C);  
};
```

➡ Tutte le istanze della classe template C, hanno come amica una ed una sola corrispondente istanza del template di classe(o funzione) friend associato.
In questo caso rimane associata come amica una ed una sola istanza del template di classe B e dei template di funzione A<T>::fun() e test.

3. Template di classe o template di funzione friend non associato (cioè tutti i suoi parametri sono tutti diversi da quelli di C) dichiarati friend all'interno del template di classe C (alcuni compilatori non supportano questa dichiarazione, g++ si dalla vers 3) :

```
template<class T>  
class C{  
    T t;  
    template<class Tp> friend int> A<Tp>::fun();  
    template<class Tp> friend class B;  
    template<class Tp> friend bool test(C<Tp>);  
};
```

➡ I template di classe(o funzione) dichiarati friend sono friend di ogni istanza del template di classe C.

Es: definizioni friend per le classi QueueItem e Queue. Dato che QueueItem non deve essere usata fuori dalla classe Queue, dichiariamo privati i suoi membri per poter poi dichiarare che Queue è una sua classe amica.

```
template<class T> class Queue; ⇒ dichiarazione incompleta
```

```
template<class T> ostream& operator<<(ostream&, const Queue<int>&); ⇒ dichiarazione incompleta dell'overloading dell'operatore di output per una coda attraverso un template di funzione esterna
```

```
template<class T>
class Queue{
public:
    friend ostream& operator<< <T>(ostream&, const Queue<T>&);
    Queue(): primo(0), ultimo(0){ }
    bool is_empty() const;
    void add(const T&);
    T remove;
    ~Queue();
private:
    QueueItem<T>* primo;
    QueueItem<T>* ultimo;
};
```

```
template<class T>
class QueueItem{
    friend class Queue<T>; ⇒ in questo modo associamo ad ogni istanza di QueueItem una sola istanza amica della classe Queue associata
    friend ostream& operator<< <T>(ostream&, const Queue<T>&);
private:
    T info;
    QueueItem* next;
    QueueItem(const T& val) : info(val), next(0) { }
};
```

```
template<class T> ostream& operator<<(ostream& os, const Queue<T>& q){
    os<<" ";
    QueueItem<T>* p = q.primo; ⇒ per amicizia con Queue
    for(; p != 0; p = p->next) os<<*p<<" "; ⇒ per amicizia con QueueItem
    os <<" " << endl;
    return os;
};
```

```
template<class T> ostream& operator<<(ostream os, const QueueItem<T>& qi){ ⇒ definizione dell'overloading dell'operatore di output
    os<<qi.info; //per amicizia con QueueItem
    return os;
}
```

//potremmo dichiarare tutte le istanze di Queue classi amiche di ogni istanza di QueueItem(cioè con Queue friend non associato di QueueItem

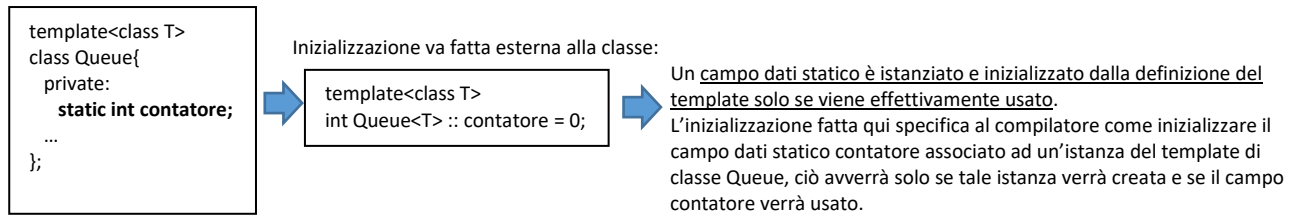
```
template<class T>
class QueueItem{
    template<class Tp> friend class Queue;
private:
    T info;
    QueueItem* next;
    QueueItem(const T& val) : info(val), next(0) { }
};
```

//Questo però non soddisfa i nostri scopi, perché non avrebbe molto senso che la classe Queue<int> sia amica di QueueItem<string>

4.3.4 Membri statici in template di classe

Anche in template di classe si possono dichiarare campi dati e metodi statici. In questo caso, ogni istanza del template di classe ha dei propri campi dati e metodi statici, distinti da quelli delle altre istanze della classe.

Es: vogliamo aggiungere al template di classe Queue un campo dati statico intero che funzioni da contatore degli oggetti QueueItem presenti nelle liste di tutti gli oggetti di una certa istanza di Queue.



4.3.5 Template di classe annidati

All'interno di un template di classe possono essere dichiarati altri template di classe annidati, sia associati che non associati.

Ad esempio, per impedire all'utente del template di classe Queue di utilizzare direttamente il template QueueItem, abbiamo dichiarato privati tutti i membri di QueueItem e quindi dichiarato Queue classe amica di QueueItem.

Un modo migliore per permettere l'uso di QueueItem solo alla classe Queue, è quello di **annidare nella parte privata della definizione del template classe Queue la definizione del template classe QueueItem:**

```
template<class T>
class Queue{
private:
    class QueueItem{ //template implicito di classe annidato associato
public:
    QueueItem(const T& val);
    T info;
    QueueItem* next;
    };
    ...
};
```

CAP 5 – Contenitori della Libreria STL

STL(standard template library) è parte della libreria standard del C++ ed è una libreria di classi contenitore, classi iteratore e algoritmi su tali classi.
E' una libreria generica, cioè i suoi membri, classi e funzioni sono dei template.

5.1 Classi Contenitore

Un contenitore è un template di classe C parametrico sul tipo T degli elementi da esso contenuti. (map e multimap sono parametrici su due tipi)
In una classe contenitore C, **una sua istanza "c" permette di memorizzare altri oggetti, cioè altri elementi di C.**

La classe contenitore C mette a disposizione vari **metodi per accedere agli elementi di una sua istanza c**

⇒ infatti esiste una **classe interna a C chiamata iteratore** i cui **oggetti possono essere usati per iterare sugli elementi delle istanze di C.**

Ogni classe contenitore C ha tra i suoi **membri** i seguenti tipi:

C::value_type: ⇒ è il tipo degli oggetti memorizzati su C. "value_type" è l'istanziatura del parametro di tipo T del template C<T>.

C::iterator: ⇒ è il tipo iteratore usato per iterare sugli elementi di un contenitore. "iterator" fornisce l'operatore di dereferenziazione operator*() che ritorna un riferimento al value_type. **(Se it è un iteratore ⇒ allora *it ha tipo value_type&)**

C::const_iterator: ⇒ è il tipo iteratore costante usato per accedere agli elementi di un contenitore che non vogliamo modificare.
(usati per accedere agli elementi di contenitori costanti)

C::size_type: ⇒ è un tipo integrale senza segno usato per rappresentare la distanza tra due iteratori.

Ogni classe contenitore C fornisce i seguenti **costruttori, metodi e operatori**:

C(const C&): ⇒ ridefinizione del costruttore di copia. Se c è costruito come copia di b ⇒ allora c contiene una copia di ogni elem di b.

C& operator=(const C&): ⇒ ridefinizione dell'assegnazione. Dopo un c = b; ⇒ il contenitore c contiene una copia di ogni elem di b.

~C(): ⇒ ridefinizione del distruttore. Nella distruzione di un contenitore c ⇒ ogni elem di c è distrutto e la memoria è deallocata.

size_type size(): ⇒ una invocazione di c.size() ⇒ ritorna la dimensione del contenitore c, cioè il numero di elementi contenuti in c.

bool empty():c.empty() ⇒ equivalente a c.size()==0.

size_type max_size():c.max_size() ⇒ ritorna la massima dimensione che il contenitore c può avere.

Per le classi contenitore vengono usati **vector, list, slist, deque, set, map, multiset, multimap**. In questi contenitori, gli elementi sono memorizzati in un ordine ben definito che non cambia da una iterazione all'altra.

Per queste classi sono disponibili:

operator== ⇒ b == c ⇒ ritorna true se b.size() == c.size() e se ogni elemento di b è uguale al corrispondente elemento di c. false altrimenti.

operator< ⇒ b < c ⇒ ritorna true se la sequenza di elementi di b è minore della sequenza degli elementi di c. (vale anche per < > >= <=)

5.2 Iteratori

Gli iteratori sono oggetti che "puntano" ad altri oggetti. Sono usati per iterare su un intervallo di oggetti.

- Se un iteratore "it" punta ad un elemento in un intervallo ⇒ allora è possibile incrementare it in modo che esso punti all'elemento successivo.
- La dereferenziazione di un iteratore it ⇒ ritorna l'oggetto puntato da it.

- Un iteratore it particolare che non punta a nessun oggetto è detto **past-the-end** se esso punta la posizione successiva dell'ultimo elemento di un contenitore.

- Un iteratore è **valido** se è dereferenzabile oppure se è past-the-end.

Ogni classe contenitore C ha 2 metodi che ritornano iteratori:

■ **C::iterator begin()** ⇒ c.begin() ⇒ ritorna un iteratore che punta al primo elemento di c.

• Se c non contiene elementi ⇒ c.begin() è l'iteratore past-the-end di c;

• Se c è costante ⇒ c.begin() ritorna un iteratore costante, di tipo C::const_iterator

■ **C::iterator end()** ⇒ c.end() ⇒ è l'iteratore past-the-end di c.

• Se c è costante ⇒ c.end() ritorna un iteratore costante.

- **C::iterator** ⇒ si usa quando serve **accedere agli L-valori del contenitore**
- **C::const_iterator** ⇒ se dobbiamo **accedere solo agli R-valori del contenitore**

Disponibili i seguenti operatori di incremento e decremento:

- **C::iterator& operator++()** ⇒ ++it ⇒ sposta l'iteratore it all'elemento successivo. (solo se it è dereferenzabile)
- **C::iterator& operator++(int)** ⇒ it++ ⇒ solito
- **C::iterator& operator--()**
- **C::iterator& operator--(int)**

5.3 Contenitori Sequenza

Sono contenitori i cui elementi sono memorizzati secondo un ordine lineare stretto, determinato dall'utente del contenitore.

Vector, list, slist, deque supportano l'inserimento e la rimozione di elementi in qualsiasi posizione di c puntata da qualche iteratore.

Dato C contenitore sequenza e una sua istanza c, si hanno le seguenti funzionalità:

- **c(n, t);** ⇒ costruisce il contenitore sequenza c contenente n copie dell'elemento t.
- **c(n);** ⇒ costruisce il contenitore sequenza c contenente n elementi inizializzati al valore di default.
- **c.insert(it,t);** ⇒ inserisce l'elemento t nella sequenza c nella posizione precedente all'elemento puntato da it e ritorna l'iteratore che punta all'elemento appena inserito.
 - **c.insert(c.begin(), t);** ⇒ inserisce t all'inizio di c.
 - **c.insert(c.end(), t);** ⇒ inserisce t in coda a c.
- **c.insert(it, n, t);** ⇒ inserisce n copie di t nella sequenza c nelle posizioni prima dell'elemento puntato da it.
- **c.erase(it);** ⇒ distrugge dalla sequenza l'elemento puntato da it e ritorna un iteratore che punta all'elemento successivo a quello rimosso.
- **c.erase(it1, it2);** ⇒ distrugge gli elementi nell'intervallo [it1,it2) e ritorna un iteratore che punta all'elem successivo all'intervallo rimosso.
- **c.clear()** ⇒ rimuove tutti gli elementi di c, è equivalente a **c.erase(c.begin(), c.end())**;

Solo per vector, list e deque:

- **c.push_back(t);** ⇒ inserisce in coda a c l'elemento t.
- **c.push_front(t);** ⇒ inserisce in testa a c l'elemento t.
- **c.pop_back();** ⇒ rimuove da c l'elemento in coda a c.
- **c.pop_front();** ⇒ rimuove l'elemento in testa a c.

5.3.1 Sequenze ad accesso casuale

vector e deque sono contenitori ad accesso casuale, ciò significa che è disponibile l'operatore di indicizzazione **operator[]**.

- Se c è il contenitore ⇒ **c[n]** ritorna l'n-esimo elemento di c dove $0 \leq n \leq n.size()$. (l'indicizzazione ha complessità media costante)

5.5 Vector

E' il più semplice ed efficiente contenitore della libreria STL.

Un vector "v" è un contenitore sequenza che supporta:

- accesso casuale agli elementi;
- rimozione e inserimento di elementi in coda a v in tempo costante;
- rimozione e inserimento di elementi in testa o a metà di v in tempo lineare sulla dimensione di v.size();

E' implementato tramite **array dinamico** ridimensionato all'occorrenza. Il numero di elementi può quindi variare dinamicamente e la gestione della memoria è automatica.

- **vector::size_type capacity() const** ⇒ v.capacity() ⇒ ritorna #elementi che v può ancora contenere senza richiedere una nuova allocaz di memoria.

Per poter dichiarare vector bisogna includere:

```
#include <vector>
```

e poi specificare il parametro attuale al template di classe vector<T> a cui sarà istanziato il tipo vector::value_type, cioè il tipo degli oggetti memorizzati.

```
#include <vector>
```

```
vector<Tipo> v; ⇒ dobbiamo specificare il parametro attuale del template di classe vector<Tipo>, cioè il tipo degli oggetti che vogliamo memorizzare  
vector<Tipo>::iterator it, it2;  
int n;
```

```
v.begin(); ⇒ ritorna l'iteratore che punta al primo elemento  
v.end(); ⇒ ritorna l'iteratore past-the-end di v  
*it; ⇒ ritorna l'elemento di v puntato da i  
it++; ⇒ it punta all'elemento successivo. Se it++ = v.end() allora it diventa l'iteratore past-the end  
++i; ⇒ come sopra  
it--; ⇒ i punta all'elemento precedente.  
--i;  
it += n; ⇒ it punta n elementi avanti  
it -= n; ⇒ it punta n elementi indietro  
j = it+n; ⇒ j punta ad it+n elementi  
it < it2; ⇒ ritorna true se it punta ad un elemento che precede nel contenitore l'elemento puntato da j  
i <= j;  
i > j;
```

Es:

```
#include<vector>
using namespace std;

template<class T>
void stampa(const vector<T>& v){  ⇒ template che stampa elementi di un vector
    for(int i = 0; i < v.size(); i++) cout<< v[i] <<endl;
}

int main(){
    const int n = 5;
    vector<int> iv(n); ⇒ dichiaro vector iv di n elem
    int ia[n] = {2,4,5,3,-2}; ⇒ dichiaro array ia di n elem
    for(int i = 0; i < n; i++) iv[i] = ia[i]+1; ⇒ inserisco elem dell'array ia sul vector iv e li aumento di 1
    vector<int> w = iv; ⇒ invocazione implicita del costruttore di copia
    vector<int> u(10, -2); ⇒ costruisco vector con 10 elementi con R-valore = -2
    u[0] = w[0]; ⇒ assegna R-valore di w[0] a u[0]
    stampa(u); ⇒ stampa 3 -2 -2 -2 -2 -2 -2 -2 -2 -2
}
```

NB: al contrario di un array, vector non si può inizializzare con una data sequenza di valori
int array[6]={1,2,3,4,5,6}; ⇒ OK
vector<int> v(6)={1,2,3,4,5,6}; ⇒ NO



Si può inizializzare un vector con un segmento di array o di un vector tramite gli opportuni costruttori:

```
int ia[20];
vector<int> iv(ia, ia+6); ⇒ costr vector con i primi 6 elem di ia
cout << iv.size(); ⇒ ritorna 6
vector<int> iv2( iv.begin(), iv.end()-2); ⇒ inizializza iv2 con i primi
cout << iv2.size(); ⇒ ritorna 4 | 4 elementi di di iv
```

Il metodo **push_back()** permette di inserire in coda un elemento:

```
vector<string> sv;
string x;
while(cin>>x) sv.push_back(x); ⇒ inserisco le stringhe x lette da cin in fondo al vector sv
cout << endl << "Abbiamo letto" << endl;
for(int i = 0; i < sv.size(); i++) cout << sv[i] << endl; ⇒ stampo tutti gli elem di sv
```

E' inoltre possibile leggere stringhe di un file (ad es "dati.txt") ed inserirle in un vector, basta che siano separate da spazi, tab o CR:

```
vector<string> sv;
string x;
ifstream file("dati.txt", ios:in);
while(file >> x) sv.push_back(x); ⇒ inserisco le stringhe x lette da cin in fondo al vector sv
vector<string> :: iterator it;
cout << endl << "Abbiamo letto" << endl;
for(it = sv.begin(); it != v.end(); it++) cout << *it << endl; ⇒ stampo tutti gli elem di vector tramite la dereferenziazione dell'iteratore it
```

CAP 6 – Ereditarietà

L'ereditarietà è uno dei concetti fondamentali della programmazione ad oggetti. Lo strumento centrale è la derivazione tra classi.

6.1 Sottoclassi

Vogliamo modellare il concetto di "orario con data" che raffina il concetto di orario. Vogliamo utilizzare la classe orario per definire una nuova classe "dataora" che eredita tutte le proprietà di orario a cui aggiungiamo ulteriori proprietà necessarie per modellare l'orario con data.

```
class orario{
public:
    orario(int o=0,int m=0,int s=0);
    int Ore() const;
    int Minuti() const;
    int Secondi() const;
    orario operator+(const orario&) const;
    bool operator==(const orario&) const;
    bool operator<(const orario&) const;
    friend ostream& operator<<(ostream&, const orario&);
private:
    int sec;
};
```

```
class dataora : public orario {
public:
    int Giorno() const;
    int Mese() const;
    int Anno() const;
private:
    int giorno;
    int mese;
    int anno;
};
```

La classe orario è una **classe base**;
La classe dataora è una **classe derivata** da orario;
La classe dataora è **sottoclasse** di orario;
La classe orario è **superclasse** di dataora;

Ogni oggetto della classe derivata dataora contiene come sottooggetto un oggetto della classe base orario.

Ciò significa che tutti i membri della classe base orario vengono implicitamente ereditati dalla classe derivata dataora, che li può liberamente usare come se fossero membri propri.

ora possiamo fare:
dataora d;
int i=d.Ore();

NB: Ogni oggetto della classe derivata è utilizzabile anche come oggetto della classe base.

Se dataora è classe derivata dalla classe base orario \Rightarrow allora c'è conversione implicita da dataora a orario che estrae da ogni oggetto x di dataora il sottooggetto x della classe base orario.

Quindi: -ogni oggetto di una classe derivata può essere convertito implicitamente in un oggetto della classe base;
-un riferimento o un puntatore ad una classe derivata può essere convertito implicitamente in riferimento o puntatore a classe base;

La classe derivata dataora è detta **sottotipo diretto** di orario, mentre orario è un **supertipo diretto** di dataora.
Per ereditarietà è possibile usare un oggetto di dataora di un tipo T, ovunque sia richiesto un oggetto di tipo T.

Relazione is-a: un oggetto di un sottotipo di T è anche tipo di T.
Perciò per un qualsiasi oggetto x di dataora \Rightarrow x ha tipo dataora che è sottotipo di orario.

Gerarchia di Classi: una classe D derivata direttamente da una classe base B può a sua volta agire da classe base per qualche altra classe E derivata direttamente da D. In questo caso si dice:
"E è sottoclasse indiretta di B, ed E è un sottotipo indiretto di B e che B è supertipo indiretto di E"

Siano X e Y due tipi qualsiasi:

- $X \leq Y \Rightarrow$ "X è un sottotipo di Y" significa che il tipo di X:
-o è uguale al tipo di Y;
-oppure che X è un sottotipo diretto o indiretto di Y.
- $X < Y \Rightarrow$ "X è un sottotipo proprio di Y" significa che X è un sottotipo di Y diverso da Y stesso.

(vale anche per puntatori e riferimenti: Se D è sottoclasse di B \Rightarrow allora il puntatore D* è sottotipo di B* e il riferimento D& è sottotipo di B&)

In ogni gerarchia di classi, le **conversioni implicite da sottotipo a supertipo** valgono per tutta la gerarchia.

Data una classe B(orario), per ogni sottotipo D(dataora) di B valgono le conversioni implicite da:

$D \Rightarrow B$ (tra oggetti, ad es dataora \Rightarrow orario)

$D\& \Rightarrow B\&$ (tra riferimenti)

$D^* \Rightarrow B^*$ (tra puntatori)

Grazie alla conversione implicita dataora \Rightarrow orario possiamo usare un oggetto dataora ovunque sia richiesto un oggetto di tipo orario:

```
int F(orario o){ ... }
dataora d;
int i=F(d); //utilizzo la funzione F che prende in input solo
l'oggetto orario interno a d di tipo dataora
```

Non vale il viceversa, dato che dataora è un oggetto di orario, mentre orario non è un dataora!

```
int G(dataora d){ ... }
orario o;
int i=G(o); //NO
```

■ **NB:** "Per supportare la programmazione object-oriented in C++, non bisogna usare oggetti, ma puntatori e riferimenti"

Significa che, se D è sottotipo di B, con b oggetto di tipo B e d oggetto di tipo D \Rightarrow con l'assegnazione b = d; si estrae "fisicamente" da d il sottooggetto di tipo B che esso contiene, ignorando il resto delle specifiche info contenute dalla sottoclasse D.
(infatti la parte di d specifica della classe D non può essere ereditata rappresentata su B)

6.1.1 Tipo Statico e Tipo Dinamico

Sia D sottoclasse di B:

```
D d;      B b;
D* pd=&d; ⇒ pd puntatore a oggetto di tipo D
B* pb=&b; ⇒ pb puntatore a oggetto di tipo B
pb=pd;    ⇒ pb diventa un puntatore a D
```

Per il puntatore pb ⇒ -Il suo **tipo statico** è **B*** (cioè il tipo determinato dal compilatore che è staticamente derivabile dal codice sorgente);
-Il suo **tipo dinamico** è **D*** dato che dopo l'assegnazione pb=pd; pb punta all'oggetto d della classe D e non più all'oggetto di classe B.

Perciò il tipo statico di un puntatore p è il tipo B* con cui è stato dichiarato, se poi durante l'esecuzione p viene fatto puntare ad un oggetto D ⇒ allora in quello stato dell'esecuzione D* diventa il tipo dinamico di p.

Il tipo dinamico di un puntatore può quindi variare a run-time. La nozione di "tipo dinamico" è un concetto astratto che si usa per ragionare logicamente sull'evoluzione dinamica del tipo degli oggetti puntati, perciò sia per il compilatore che per il sistema che gestisce l'esecuzione del programma, ogni puntatore ha solo e solamente il tipo con cui è stato dichiarato.

Analogo per i riferimenti:

```
D d; B b;
D& rd=d;
B& rb=d;
```

Per il riferimento rb ⇒ - il suo tipo statico è B&;
- il suo tipo dinamico è D&, dato che con rb = d; viene definito rb come alias dell'oggetto d.

Notaz: **TS(p)**= tipo statico del puntatore p, **TD(d)**=tipo dinamico di p
TS(r)= tipo statico del riferimento r, **TD(r)**=tipo dinamico di r

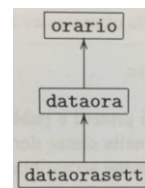
6.1.2 Gerarchia di Classi

Come detto una classe derivata può a sua volta essere usata come classe base per un ulteriore passo di derivazione.

Se vogliamo definire un tipo che oltre alle proprietà di dataora, memorizzi anche il giorno della settimana:

```
enum settimana{lun,mar,mer,gio,ven,dom};
class dataorasett: public dataora{
public:
    settimana GiornataSettimana() const;
private:
    settimana giornosettimana;
};
```

In questo modo abbiamo una gerarchia a 3 classi con il seguente diagramma della gerarchia:



6.1.3 Accessibilità

Una classe derivata non ha accesso alla parte privata della sua classe base.

La parte privata di una qualsiasi classe B(orario) è inaccessibile alle classi derivate da B(dataora).

Se vogliamo aggiungere alla classe dataora un metodo set2000() che assegna all'oggetto di invocazione le ore 00:00:00 del 1 gen 2000, dobbiamo dichiarare **protected** il campo privato sec di orario(invece che private) in modo che i membri al suo interno risultino accessibili alle classi derivate da B e sempre non accessibili alle classi esterne a orario.

```
class orario{
    protected:
        int sec; //ora sec è accessibile a set2000()
    ...
};
```

```
dataora::set2000(){
    sec=0;
    giorno=1;
    mese=1;
    anno=2000;
}
```

protected vs private:

- protected**: membri e metodi di una classe base sono accessibili e modificabili solo dalle classi derivate che li ereditano, mentre all'esterno della classe base o delle sue derivate non sarà possibile accedervi.
- private**: le proprietà dichiarate private sono accessibili e modificabili solo dalla classe che li dichiara.

6.1.3.1 Tipologie di derivazione e livello di accessibilità dei membri ereditati

Derivazione pubblica: la classe dataora è ottenuta dalla classe orario tramite la derivazione pubblica

```
class dataora: public orario{ ...}
```

E' anche detta ereditarietà di tipo ed è la forma di derivazione più diffusa.

In questo modo i **campi protetti e pubblici della classe base**(orario) **rimangono pubblici anche nella classe derivata**(dataora).

(Se ho orario⇒dataora⇒dataorasett, allora anche tutte le altre classi derivate da orario avranno accesso al campo privato sec)

Questo permette di realizzare la relazione di sottotipo "is-a" dato che l'interfaccia pubblica di una classe D derivata direttamente da una classe base B si ottiene tramite l'interfaccia pubblica originaria di B, a cui si aggiungono i nuovi membri della classe derivata D.

Derivazione privata: rende privati tutti i membri protetti e pubblici della classe base alle classi derivate

```
class dataora: private orario{ ...}
```

E' anche detta ereditarietà di implementazione.

In questo caso l'interfaccia pubblica della classe base(orario) non è inclusa nell'interfaccia pubblica della classe derivata(dataora), dato che la parte pubblica della classe base non è accessibile dalle classi esterne ad essa in quanto dichiarata privata.

(Se ho orario⇒dataora⇒dataorasett, allora tutte le classi derivate da orario non avranno accesso ai campi di orario)

Derivazione protetta: rende protetti nella classe privata i membri pubblici e protetti della classe base

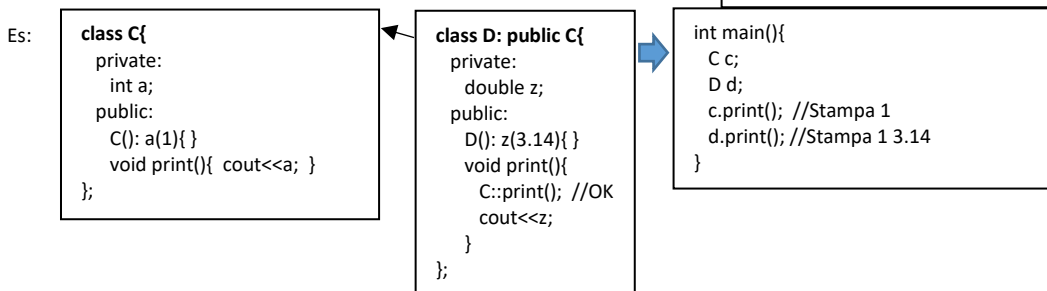
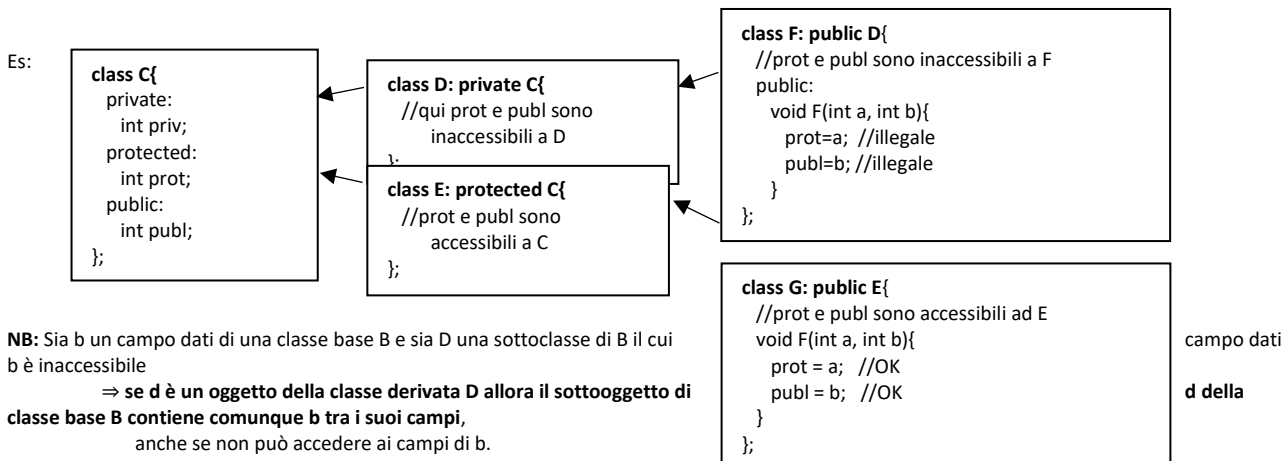
```
class dataora: protected orario{ ...}
```

Le ultime due non hanno effetti sui membri privati della classe base, dato che sono inaccessibili per la classe derivata e tutte le altre classi esterne.

Derivazione		public	protected	private
Membro				
private		inaccessibile	inaccessibile	inaccessibile
protected		protetto	protetto	privato
public		pubblico	protetto	privato

NB: Le conversioni implicite indotte dalla derivazione valgono solo per la derivazione pubblica (relazione is-a), mentre per la derivazione protetta e privata non ci sarà alcuna conversione implicita.

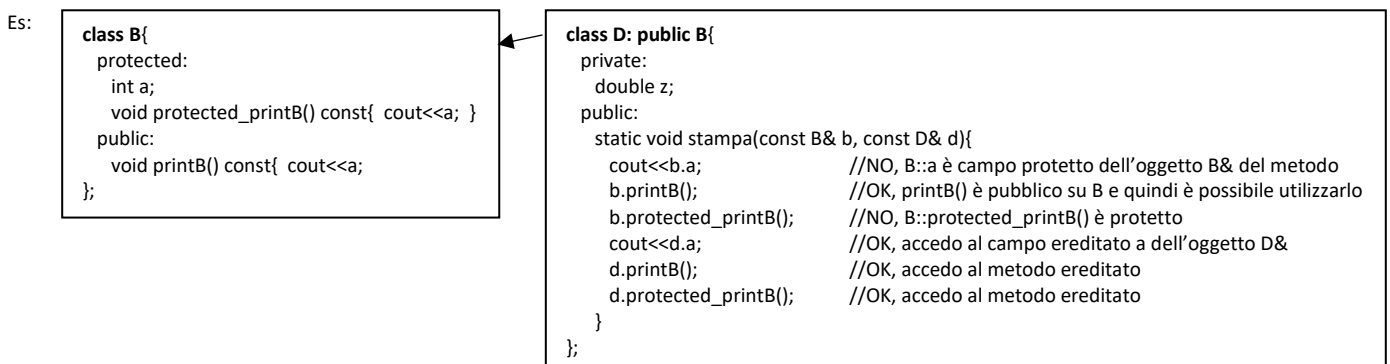
NB: Usare `protected` va contro il principio dell'incapsulamento dei dati e dell'information hiding in quanto la modifica di un membro protetto di una classe base potrebbe richiedere la successiva modifica di tutte le classi derivate. L'attributo `protected` va quindi usato con attenzione, in generale bisogna quindi progettare classi con campi privati.



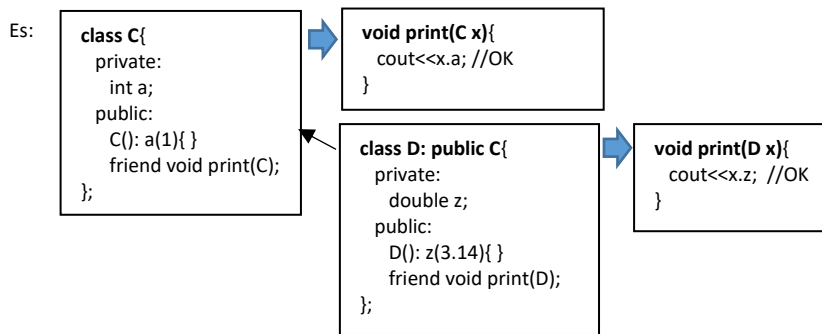
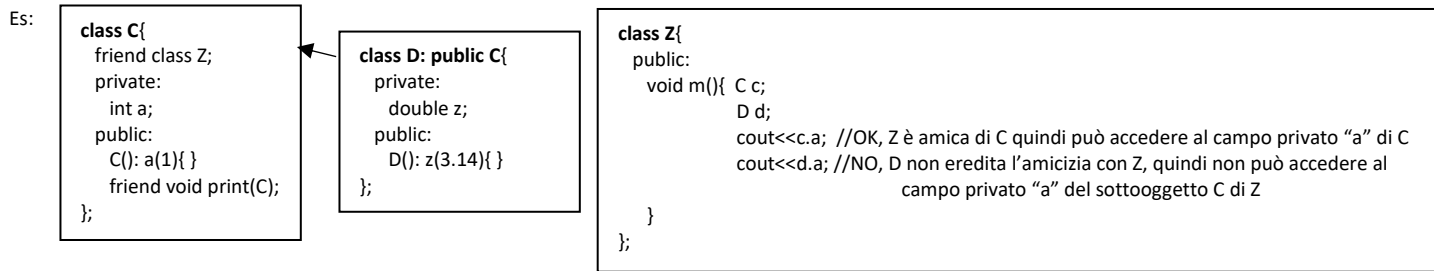
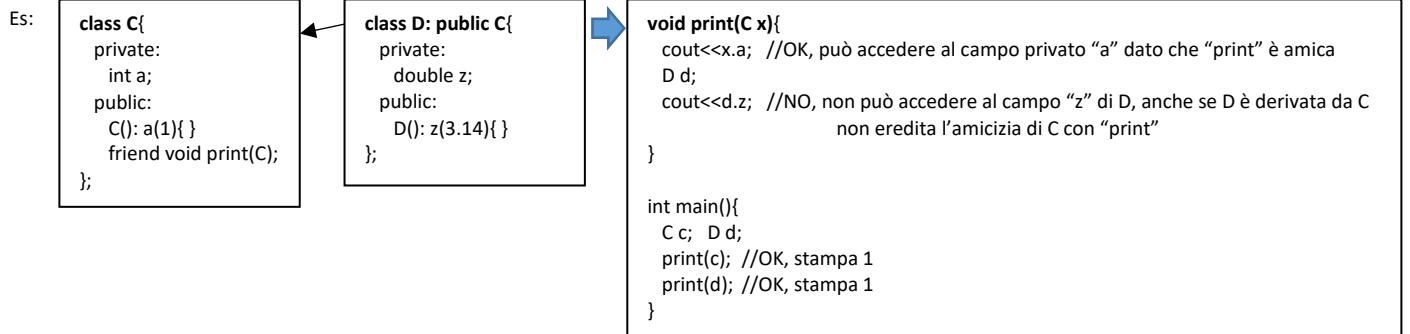
NB: Sia `b` membro di una classe base `B` e sia la classe derivata `D` una sottoclasse di `B` che eredita `b` come membro protetto
 ⇒ Ciò permette alla classe `D` di accedere al membro `b` dei sottooggetti di tipo `B` degli oggetti appartenenti alla classe `D`, ma non permette alla classe `D` di accedere al membro `b` degli oggetti che invece appartengono alla classe base `B`.

NB: Sia il membro `b` di `B` un campo dati, se nel corpo di un metodo `T F(B x,...)` della classe `D` c'è un parametro di tipo `B`
 ⇒ allora non si avrà accesso al campo dati `x.b` dato che il membro `b` di `B` è inaccessibile in quanto tento di accedere ad un oggetto di `B`.

NB: Sia il membro `b` di `B` un campo dati, se nel corpo di un metodo `T F(D x,...)` della classe `D` c'è un parametro di tipo `D`
 ⇒ allora si avrà accesso al campo dati `x.b` in quanto `b` è ereditato in `D` come campo dati protetto.



NB: Data una classe base `B` con una classe `D` sottoclasse di `B`, siano una funzione esterna `F` e una classe esterna `C` dichiarate friend di `B`
 ⇒ allora tra `D`, `F`, `C` non esiste nessun rapporto, in quanto `D` non eredita le amicizie di `B` che esse siano funzioni o classi amiche.



6.1.4 Conversioni esplicite

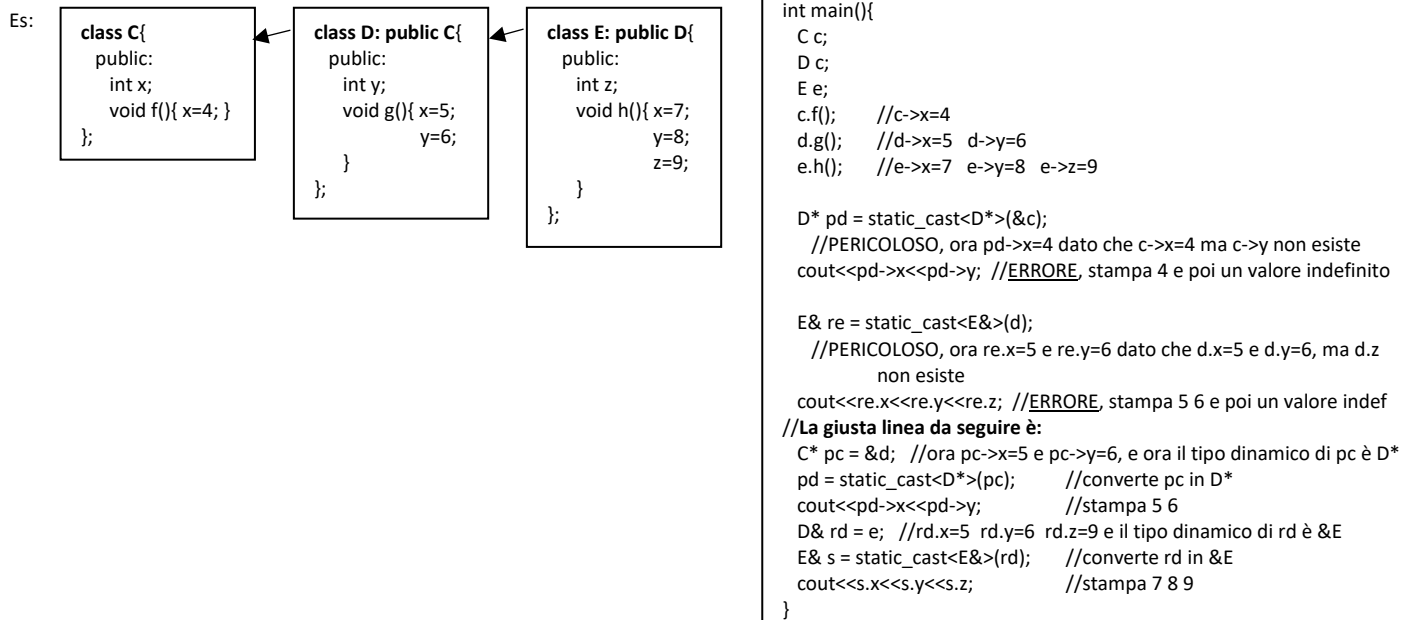
$B^* \Rightarrow D^*$
 $B\& \Rightarrow D\&$

E' sempre possibile effettuare una conversione esplicita tramite `static_cast` per convertire un puntatore/riferimento di una classe base B in un puntatore/riferimento ad una classe D derivata da B.

Si deve quindi garantire la correttezza di queste conversioni e prima di effettuarle si dovrà avere la certezza che il tipo dinamico del puntatore o del riferimento che vogliamo convertire sia E* oppure E& per un qualche tipo E sottotipo di D.

Ad esempio, dato un puntatore "pb" di tipo B*, se voglio convertirlo al tipo D*, il tipo dinamico di pb immediatamente prima della conversione esplicita dovrà essere E* per un tipo E sottotipo di D.

Se ciò non è verificato, il codice potrebbe avere un comportamento non prevedibile e potrebbe causare errori.



6.2 Ridefinizione metodi e campi dati

Ad una classe D derivata da B, si aggiungono dei membri propri (campi dati, metodi, classi annidate etc) ai membri ereditati dalla classe base B.

In D è quindi possibile: **-ridefinire i campi dati e i metodi ereditati da B.**

-usare l'operatore di scoping B::b per accedere al membro b definito in B.

Es: vogliamo ridefinire l'operatore di somma della classe dataora, ridefinendo quindi operator+ ereditato dalla classe orario.

```
dataora dataora::operator+(const orario& o) const{
    dataora aux = *this;
    aux.sec = sec + 3600*o.Ore() + 60*o.Minuti() + o.Secondi();
    if(aux.sec >= 86499){
        aux.sec = aux.sec - 86400;
        aux.AvanzaUnGiorno();
    }
    return aux;
}
```

In questo caso sommiamo l'oggetto di invocazione di tipo dataora con un parametro di tipo orario, ritornando poi un oggetto di tipo dataora.

(NB: se avessimo fatto aux.sec = sec + o.sec; avrebbe dato errore dato che sec è privato in orario)

```
dataora::AvanzaUnGiorno(){
    if(giorno < GiorniDelMese()) giorno++;
    else if(mese < 12){ giorno = 1;
        mese++;
    }else{ giorno = 1;
        mese = 1;
        anno++;
    }
}
```

```
orario o1, o2;
dataora d1, d2;
o1 + o2;           //invoca orario::operator+
d1 + d2;           //invoca dataora::operator+
o1 + d2;           //invoca orario::operator+
d1 + o2;           //invoca dataora::operator+
dataora x = o1 + o2; //ERRORE, nessuna conversione orario⇒dataora
orario y = d1 + d2;  //OK
d1.orario::operator+(d2); //invoca orario::operator++
```

NB: Se si vuole ridefinire un campo dati b di una classe B in una classe derivata D (solo se b è accessibile a D) è possibile farlo definendo nella classe derivata D un nuovo campo dati con lo stesso identificatore "b", il cui tipo potrà essere diverso da quello di b in B. (non molto usato)

Es:

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

```
class D: public B{
private:
    double x; //ridefinizione del campo dati x
public:
    D(): x(3.14){ }
    void print(){ cout<<x; } //si riferisca alla x di D
    void printAll(){ cout<<B::x<<x; }
};
```

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

Name Hiding Rule: sia D una classe derivata dalla classe base B, sia m() un metodo di B accessibile in D e possibilmente sovraccaricato

⇒ allora la ridefinizione in D del nome del metodo m() nasconde sempre tutte le versioni sovraccaricate di m() disponibili in B, che quindi non saranno direttamente accessibili in D ma solo tramite l'operatore di scoping B::

Ciò vale per ogni ridefinizione di m() che può essere di 3 tipologie:

- (1) Stessa segnatura (lista parametri e tipi ritorno) di una delle versioni di m() disponibili in B;
- (2) Stessa lista dei parametri ma diverso tipo di ritorno;
- (3) Diversa lista dei parametri e stesso tipo di ritorno.

NB: Overloading ≠ Name Hiding Rule

Quindi se in dataora() ridefiniamo il metodo Ore con la nuova segnatura

```
int dataora::Ore(int) const
```

```
dataora d;
cout<<d.Ore();           //NO
cout<<d.orario::Ore(); //OK
```

```
oppure con la dichiarazione d'uso
using dataora::Ore;
cout<<d.Ore();           //OK
```

L'effetto della richiarazione d'uso è quello di inserire il membro orario::Ore() della classe base orario nell'insieme dei metodi sovraccaricati associati al nome del metodo Ore della classe derivata dataora.

NB: Una dichiarazione d'uso per un metodo specifica solo l'identificatore del metodo e non la lista dei parametri.

Se il metodo m() oggetto della dichiarazione d'uso nella classe D derivata da B è sovraccaricato all'interno di B

⇒ allora tutti gli overloading di m() in B verranno ereditati in D.

Es:

```
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)"; }
};
```

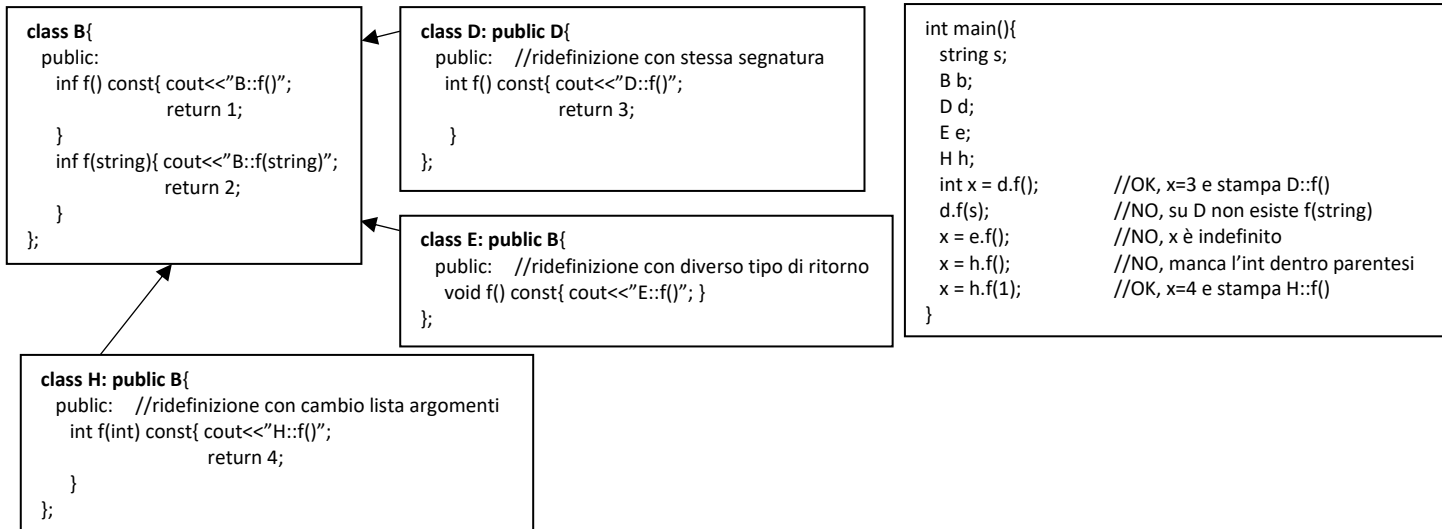
```
class D: public B{
public:
    using B::m; //dichiarazione d'uso
    //overloading di m
    void m(int x){ cout<<"D::m(int)"; }
    void m(){ cout<<"D::m()"; }
};
```

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

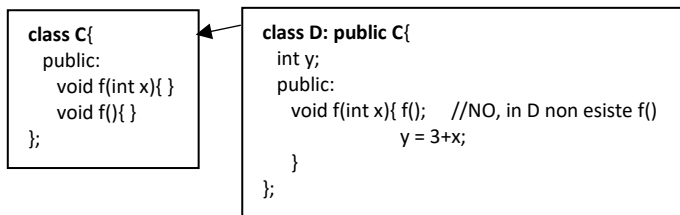
NB: Sia D sottoclasse di B con m membro di B (campo dati o metodo) accessibile in D. Sia B* pb puntatore alla classe base definito in un contesto in cui pb ha accesso al membro m \Rightarrow allora pb \rightarrow m seleziona sempre il membro m della classe base B e non il membro definito in D, anche quando il tipo dinamico di pb è D*. (lo stesso per un riferimento con rb.m)

Ciò avviene perché il legame tra oggetto di invocazione e metodo invocato m() è statico, cioè determinato dal compilatore che considera il tipo statico del puntatore/riferimento.

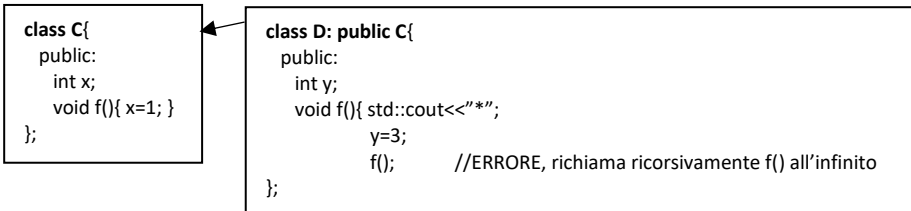
Es:



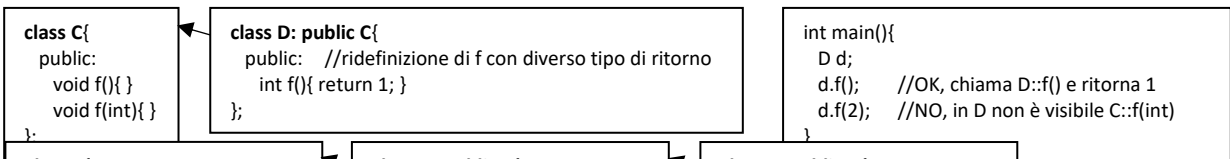
Es:



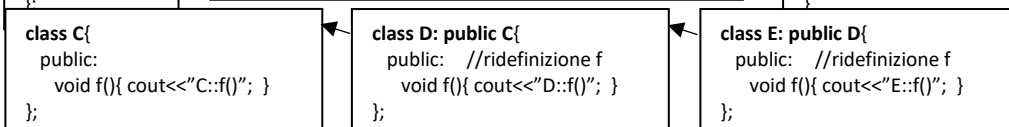
Es:



Es:



Es:



```

int main(){
    C c; D d; E e;
    C* pc = &c;
    E* pe = &e;
    c = d; //Conversione D  $\Rightarrow$  C
    c = e; //Conversione E  $\Rightarrow$  C
    d = e; //Conversione E  $\Rightarrow$  D
    d = c; //NO, C non ha classe base D
    C &rc = d; //Conversione C  $\Rightarrow$  D
    D& rd = e; //Conversione D  $\Rightarrow$  E
    pc->f(); //stampa C::f()
    pc = pe; //Conversione C*  $\Rightarrow$  E*
    rd.f(); //stampa D::f() per l'NB sopra
    c.f(); //stampa C::f() ""
    pc->f(); //stampa C::f() ""
}

```

Es:

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

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

```
int main(){
    int n;
    double x;
    int* q;
    D d;
    d.i = n;    //OK, stesso tipo int
    d.a = q;    //OK, stesso tipo int*
    d.a = x;    //NO, tipo diverso int* e double
    d.C::a = x; //OK, stesso tipo double
    d.f(q);     //OK, stampa D::f(int*)
    d.f(x);     //NO, tipo parametro ≠ da int*
    d.C::f(x);  //OK, tipo parametro = a double
}
```

Es:

```
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;    //stampa 1
    cout<<d.x<<d.y; //stampa 1|2
}
```

Es:

```
class C{
public:
    int a;
    void fC(){ a=2; }
};
```

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

```
class E: public D{
public:
    char a;
    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;    //stampa 3.14|*
    cout<< pd->a << pd->D::a << pd->C::a; //stampa 3.14|3.14|4
    cout<< pe.a << pe.D::a << pe.C::a; //stampa *|6.28|5
    cout<< e.a << e.D::a << e.C::a;    //stampa *|6.28|5
}
```

6.3.1 Costruttore nelle classi derivate

Ovviamente non vengono ereditati dalla classe derivata, ma c'è la possibilità di invocare quelli della classe base in quelli della classe derivata.

Data la classe D derivata dalla classe base B, quando istanziamo un oggetto d di D, bisognerà richiamare nel costruttore di D un costruttore di B, implicitamente o esplicitamente, per creare ed inizializzare il sottooggetto di d della classe base B.

Invocazione Esplicita: è possibile inserire nella lista di inizializzazione del costruttore di D un'invocazione esplicita dei costruttori di B;

Invocazione Implicita: se la lista di inizializzazione del costruttore di D non include invocazioni esplicite di costruttori di B, allora viene automaticamente invocato il costruttore di default di B (che dovrà quindi essere disponibile).

NB: È importante non confondere costruttori e campi dati della classe base B:

⇒ la lista di inizializzazione di D non può contenere invocazioni di costruttori per i campi dati della classe base B.

NB: Nell'esecuzione di un costruttore per D:

1. Viene invocato sempre per primo il costruttore della classe base B da cui deriva, esplicitamente o implicitamente;
2. Viene eseguito il costruttore per i campi dati propri di D;
3. Viene eseguito il corpo del costruttore di D.

(Se su D non mettiamo alcun costruttore, interverrà il costruttore di default standard di D)

Es:

```
dataora d;  
cout<<d.Ore(); //stampa 0  
cout<<d.Giorno(); //stampa valore indefinito
```

➡ L'oggetto "d" viene costruito dal costruttore di default standard di dataora, che prima richiama il costruttore standard per il sottooggetto della classe base orario che assegna sec=0; e poi richiama i costruttori di default di giorno, mese, ora di tipo int i quali avranno valori indefiniti.

Se invece definiamo noi un costruttore di default:

```
dataora::dataora(): giorno(1), mese(1), anno(2000) { }  
dataora d;  
cout<<d.Ore(); //stampa 0  
cout<<d.Giorno(); //stampa 1
```

Possiamo inoltre definire un costruttore che inizializzi tutti i campi dati di dataora, sia quelli propri che quelli della classe orario:

```
dataora::dataora(int a, int me, int g, int o, int m, int s):  
    orario(o,m,s), giorno(g), mese(me), anno(a){ }  
dataora d(2003,11,17,11,55,13);  
cout<<d.Ore(); //stampa 11  
cout<<d.Giorno(); //stampa 17
```

Lo stesso può essere fatto con il costruttore di copia, assegnazione e distruttore.

Es:

```
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; //stampa C01|Z0  
} //prima costruttore della classe base C e  
    poi quello dei campi di D
```

Es:

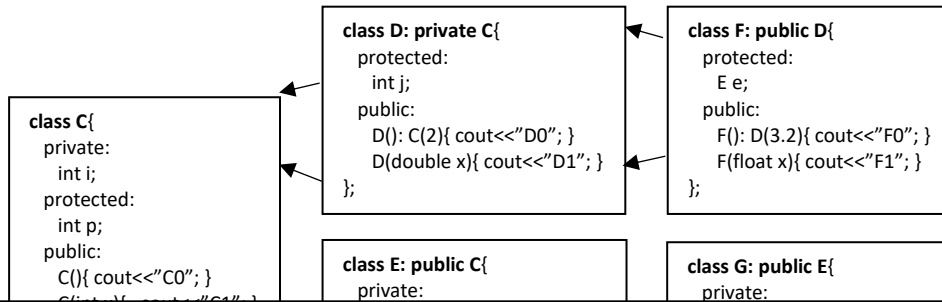
```
class Z{  
public:  
    Z(){ cout<<"Z0"; }  
    Z(double d){ cout<<"Z1"; }  
};
```

```
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; //stampa Z1|8|C0|Z0|D0  
    1. Partendo da D, viene preso in considerazione il costruttore D() in non è specificato alcun costruttore per la classe base C, perciò  
        ⇒ 1. viene richiamato il costruttore standard C() ⇒ 2. con w(6.28) richiama il costruttore della classe Z(double) ⇒ stampa Z1;  
        3. poi assegna x(8) ⇒ stampa 8|C0  
    3. Ritorna in D e inizializza i campi propri di D ⇒ con Z z; richiama il costruttore standard Z() (dato che non è specificato sul costruttore di D) ⇒ stampa Z0  
    4. Alla fine esegue il corpo del costruttore di D ⇒ stampa D0  
    D e(4); //stampa Z0|4|C1|Z1|D1  
    1. Partendo da D, viene preso in considerazione il costruttore D(int) di D dove viene specificato il costruttore per la classe base C(int), perciò  
        ⇒ 1. viene richiamato il costruttore C(int) ⇒ 1. inizializza il campo proprio Z w; ⇒ stampa Z0  
        2. assegna x(z) ⇒ stampa 4|C1  
    2. Ritorna in D e inizializza Z z; con z(3.14) ⇒ viene richiamato il costruttore Z(double) ⇒ stampa Z1  
    3. Alla fine esegue il corpo del costruttore di D ⇒ stampa D1  
}
```


Es:



```
int main(){
    G g;    //stampa C1|C1|D0|E0|C0|D1| C1|C1|D0|E0|F0|C0|G0
    1. Partendo da G, nel suo costruttore G() è specificato il costruttore per la classe base da cui deriva E
        ⇒ 1. Nel costruttore di E è specificato il costruttore C(int) per la classe base C da cui deriva
            ⇒ 1. Viene eseguito il corpo del costruttore della classe base C(int) ⇒ stampa C1
        2. Ritorna ad E e dove viene inizializzato il campo proprio D d;
            ⇒ 1. Vado su D dove nel costruttore D() è specificato il costruttore per la classe base C da cui deriva
                ⇒ 1. Viene eseguito il corpo del costruttore C(int) ⇒ stampa C1
                2. Ritorna a D dove esegue il corpo del costruttore D() ⇒ stampa D0
            3. Ritorna ad E dove viene eseguito il corpo del costruttore E() ⇒ stampa E0
    2. Ritorna a G dove vengono inizializzati i campi propri
        ⇒ 1. F f; ⇒ 1. Nel costruttore di F è specificato il costruttore D(double) per la classe base D da cui deriva
            ⇒ 1. Nel costruttore D(double) non è specificato il costruttore per la classe base C da cui deriva
                ⇒ 1. Viene eseguito il costruttore C() per la classe base C ⇒ stampa C0
                2. Ritorna in D dove esegue il corpo del costruttore D(double) ⇒ stampa D1
            2. Ritorna ad F ed inizializza il campo proprio E e;
                ⇒ 1. Vado su E dove sul costruttore E() è specificato il costruttore C(int) per la classe base C da cui deriva
                    ⇒ 2. Esegue il costruttore C(int) ⇒ stampa C1
                2. Ritorno su E dove inizializza il campo proprio D d;
                    ⇒ 1. Vado su D dove nel costruttore D() è specificato il costruttore per la classe base C da cui deriva
                        ⇒ 1. Viene eseguito il corpo del costruttore C(int) ⇒ stampa C1
                        2. Ritorna a D dove esegue il corpo del costruttore D() ⇒ stampa D0
                    3. Ritorna ad E dove viene eseguito il corpo del costruttore E() ⇒ stampa E0
                3. Ritorna a F ed esegue il corpo del costruttore F() ⇒ stampa F0
            ⇒ 2. C c; ⇒ 1. Esegue il corpo del costruttore C() ⇒ stampa C0
    3. Esegue il corpo del costruttore G() ⇒ stampa G0
}
```

6.3.2 Costruttore di Copia nelle classi derivate

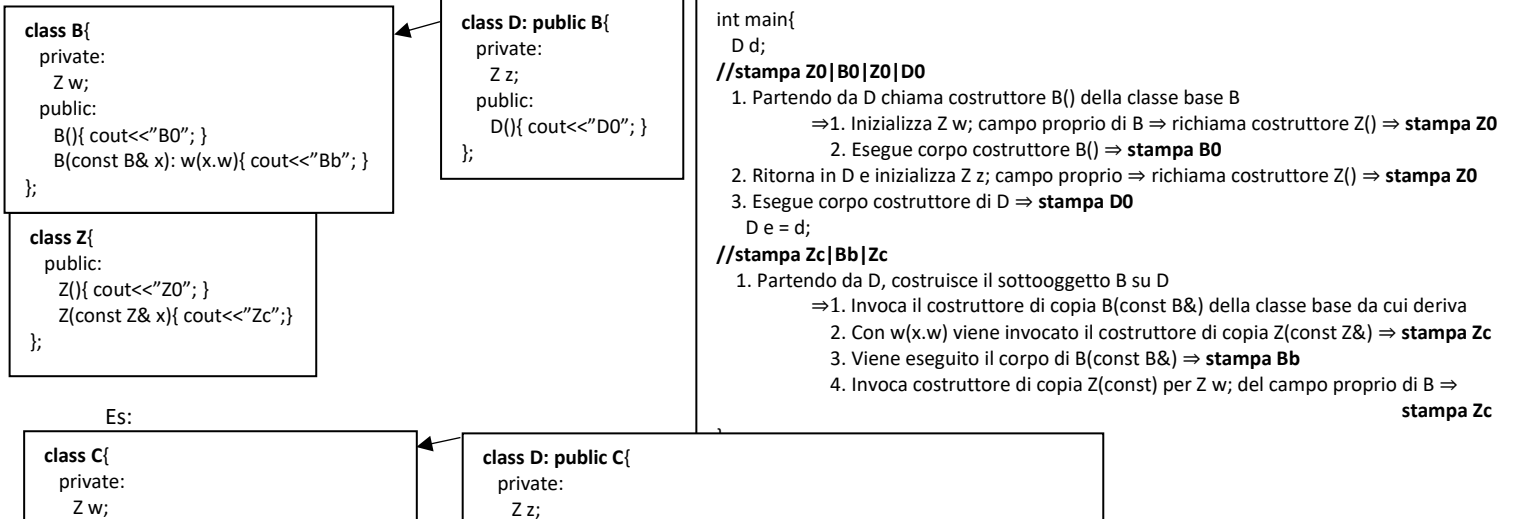
NB: Il **costruttore di copia standard** di una classe D derivata direttamente da una classe base B invocato su un oggetto x di D :

- ⇒ 1. Costruisce il sottooggetto di B invocando il costruttore di copia B (const B&) di B (standard o ridefinito) sul corrispondente sottooggetto x;
- 2. Successivamente costruisce ordinatamente i campi dati propri di D invocando i relativi costruttori di copia.

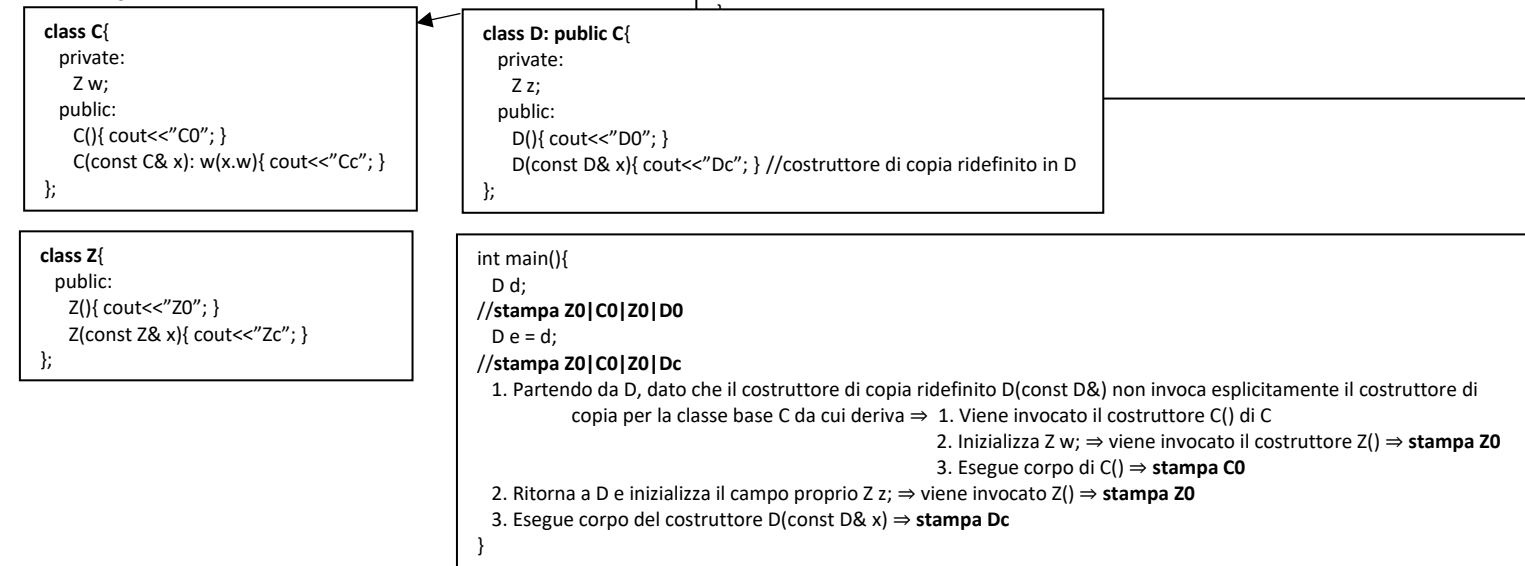
NB: Se il costruttore di copia viene ridefinito in D, esso può invocare esplicitamente il costruttore di copia di B o qualsiasi altro costruttore di B

Se non ci sono invocazioni esplicite per il costruttore di copia di B, viene invocato automaticamente il costruttore di default di B (QUINDI NON IL COSTRUTTORE DI COPIA)

Es:



Es:



6.3.3 Assegnazione nelle classi derivate

NB: L'assegnazione standard di una classe D derivata direttamente da una classe base B:

1. Invoca l'assegnazione della classe base B (standard o ridefinita) sul sottooggetto corrispondente;
2. Successivamente esegue l'assegnazione ordinatamente membro per membro dei campi dati propri di D invocando le corrispondenti assegnazioni (standard o ridefinite);

NB: Se l'assegnazione viene ridefinita in D \Rightarrow viene eseguito solo il suo corpo, la ridefinizione dell'assegnazione non provoca alcuna invocazione implicita.

Es:

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

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

```
int main(){
    D d;    //stampa Z0|C0|Z0|D0
    D e;    //stampa Z0|C0|Z0|D0
    e = d;   //stampa Z=|C=|Z=
            1. Partendo da D viene invocata l'assegnazione C& operator=(const C&) della classe C
            da cui deriva  $\Rightarrow$  1. Con w=x.w; viene invocata l'assegnazione di Z  $\Rightarrow$  stampa Z=
            2. stampa C=
            2. Ritorna in D e invoca l'assegnazione di Z per il campo proprio Z z;  $\Rightarrow$  stampa Z=
}
```

Es:

```
class C{
public:
    Z w;
    C(){ cout<<"C0"; }
    C(const Z& x){ cout<<"Zc"; }
    C& operator=(const Z& x){ cout<<"Z="; 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; }
};
```

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

```
int main(){
    D d;    //stampa Z0|C0|Z0|D0
    d.w.x = 3;
    D e;    //stampa Z0|C0|Z0|D0
    e.w.x = 5;
    e = d;   //stampa Z=|D=
            1. Esegue il corpo di D  $\Rightarrow$  1. con z=x.z; invoca l'assegnazione di Z  $\Rightarrow$  stampa Z=
            2. stampa D=
    cout<<e.w.x<<d.w.x; //stampa 5|3 dato che sulla ridefinizione dell'assegnazione di D non è
                        specificata nessuna assegnazione per il campo w di C.
}
```

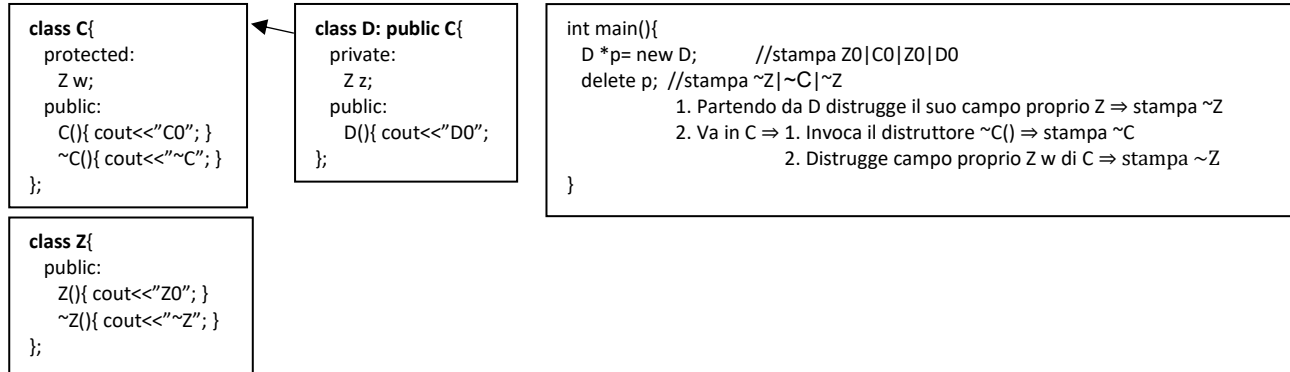
6.3.2 Distruttore nelle classi derivate

NB: Il **distruttore standard** di una classe D derivata direttamente da B:

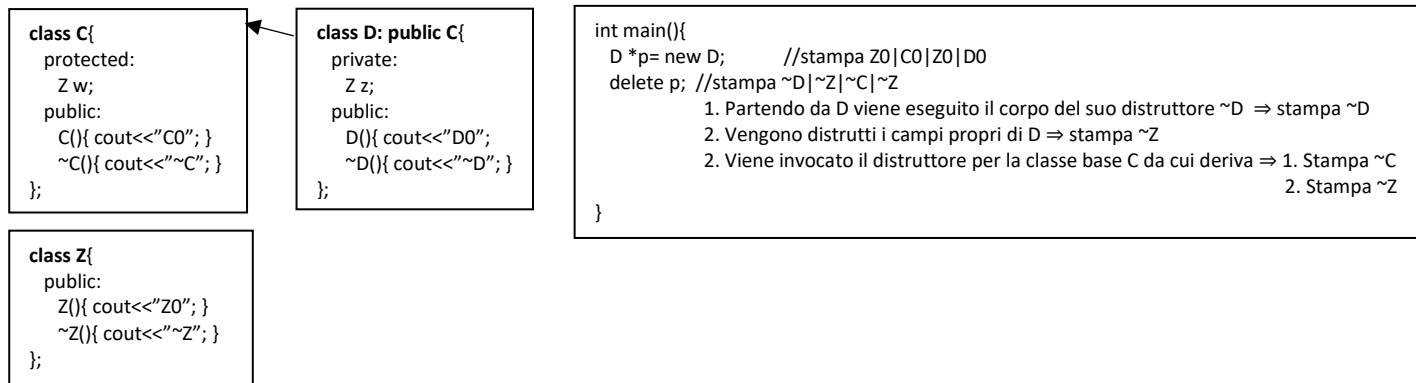
1. Invoca il distruttore standard proprio di D \Rightarrow distrugge i campi dati propri di D nell'ordine inverso a quello di costruzione tramite i corrispondenti distruttori (standard o ridefiniti);
2. Invoca implicitamente il distruttore della classe B da cui deriva (standard o ridefinito) per distruggere il sottooggetto B;

NB: Se il distruttore viene ridefinito in D \Rightarrow 1. Viene eseguito il corpo del distruttore di D;
2. Viene invocato il distruttore della classe base B (standard o ridefinito) per distruggere il sottooggetto di B;

Es:



Es:



6.4 Ereditarietà e Template

L' ereditarietà può essere usata anche con i template di classe e sia la classe base che la classe derivata possono essere definite come template di classe.

Le 3 modalità di derivazione per i template di classe sono:

(A) Classe base template e classe derivata da una istanza della classe base:

```
template <class T>
class Base{ ... };
class Derivata : public base<int>{ ... };
```



Qui ogni oggetto della classe Derivata contiene come sottooggetto un oggetto dell'istanza base<int> della classe template Base.

(B) Classe base non template e classe derivata template:

```
class Base{ ... };
template<class T>
class Derivata : public base{ ... };
```



Qui ogni oggetto di ogni istanza della classe template Derivata contiene come sottooggetto un oggetto della classe base Base.

(C) Classe base e classe derivata entrambe template:

```
template <class T>
class Base{ ... };
template<class Tp>
class Derivata : public base<Tp>{ ...
};
```



Detta **Derivazione Associata**: i parametri di tipo(o di valore) della classe template Derivata devono essere un sovrainsieme di quelli della classe Base.
Qui ogni oggetto di una istanza della classe template Derivata contiene come sottooggetto un oggetto dell'istanza associata della classe template Base.

NB: Nella (C) bisogna stare attenti alle modalità di accesso ai membri della classe Base, infatti **per accedere ad un membro m** (che sia un campo dati o un metodo) **della classe base tramite l'oggetto di invocazione è necessario usare esplicitamente la sintassi**

this->m

Es:

```
template<class T> class B{
public:
    int m;
    int n;
    int f(){ ... };
    int g(){ ... };
};

int n=1; //variabile globale
int g(){ ... };//funzione globale
```



```
template<class T> class D : public B<T>{
public:
    void h(){m = 2; //NO
        this -> m = 2; // OK
        f(); //NO
        this -> f(); //OK
        n = 3; //OK, assegnazione sulla variabile globale ::n
        this -> n= 3; //OK, assegnazione al campo dati B::n
        g(); //OK, invoca funzione globale ::g()
        this -> g(); //OK, invoca il metodo B::g()
    }
};
```

6.5 Metodi Virtuali

Un oggetto di una classe derivata può essere usato ovunque sia richiesto un oggetto della classe base, in quanto esiste una conversione implicita da oggetti di una classe derivata ad oggetti di una classe base, che estrae i corrispondenti sottooggetti.

```
void F(orario o);  
dataora d;  
F(d);
```

→ In questa funzione, il parametro "o" di tipo orario viene passato per valore

→ con l'invocazione di F ⇒ viene invocato il costruttore di copia di orario, che costruisce il parametro formale "o" copiando in esso solo il sottooggetto di tipo orario del parametro attuale d, ignorando tutto il resto.

```
void G(const orario& o);  
dataora d;  
G(d);
```

→ il parametro "o" di tipo orario viene passato per riferimento costante

→ con l'invocazione di G ⇒ al contrario del passaggio per valore, **non viene effettuata una copia di d, ma il parametro formale diventa un alias dell'oggetto d di tipo dataora.**

Nella funzione G abbiamo che TS(o)=const orario& ⇒ quindi nel corpo di G, il parametro "o" viene usato come sottooggetto di tipo orario dell'oggetto "d" di dataora.

Supponiamo che in G, il parametro "o" sia usato come oggetto di invocazione di un metodo Stampa()

```
void G(const orario& o){  
    o.Stampa();  
}  
dataora d;  
G(d);
```

→ In questo caso tra l'oggetto di invocazione "o" e la funzione Stampa() c'è un **legame statico** e viene quindi **invocato il metodo orario::Stampa() e non dataora::Stampa()**.

Quando il parametro viene passato per riferimento, si può fare in modo che l'associazione tra oggetto di invocazione e metodo da invocare venga **effettuata a tempo di esecuzione, in base al tipo dinamico del parametro** e non in base al suo tipo statico.

Ciò si può fare dichiarando un metodo virtuale:

```
class orario{  
    virtual void Stampa();  
    ...  
};
```

```
void G(const orario& o){  
    o.Stampa();  
}
```

→ In questo caso verrà invocato: -dataora::Stampa() se TD(o)=const dataora&
-orario::Stampa() se TD(o)=const orario&

Perciò, in una invocazione di G(x) ⇒ -Se il parametro attuale x è di tipo dataora viene invocato dataora::Stampa();
-Se x è di tipo orario viene invocato orario::Stampa();

In questo caso c'è **legame dinamico**(dynamic binding) tra oggetto di invocazione e metodo virtuale.

Il **metodo virtuale da invocare effettivamente verrà quindi selezionato solamente a tempo di esecuzione** e non staticamente dal compilatore.

Questo oltre che per i riferimenti vale **anche per i puntatori**:

```
dataora d;  
orario* p=&d;  
p->Stampa(); //oppure (p*).Stampa();
```

→ In questo caso: -se orario::Stampa() era stato dichiarato virtuale ⇒ viene invocato dataora::Stampa();
-altrimenti se non è virtuale viene staticamente determinato orario::Stampa().

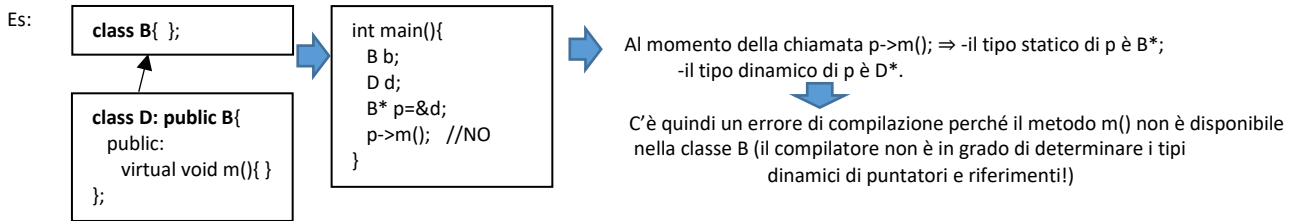
Overriding di m() è la ridefinizione di un metodo virtuale m() ⇒ Marcando un metodo m() come virtuale di una classe B, il progettista **delega alle ridefinizioni di m() delle sottoclassi di B il compito di implementare il metodo m() in modo specifico alla particolare sottoclasse.**

NB: Quando in una classe B un suo metodo m() viene dichiarato virtuale ⇒ allora m() resta virtuale anche in tutta la gerarchia di classi che derivano da B, anche se m() non è dichiarato esplicitamente virtuale nella ridefinizione operata dalle sottoclassi di B.

(Per maggior chiarezza è meglio dichiarare esplicitamente che un metodo è virtuale in ogni sottoclasse in cui è definito)

NB: Ciò si applica **anche agli operatori, che possono essere ridefiniti dichiarandoli virtuali**. In questo caso diventano metodi virtuali a tutti gli effetti.

NB: Affinchè una invocazione del metodo virtuale `m()` tramite un puntatore `p` di tipo `B*` possa compilare, è necessario che il metodo `m()` sia disponibile nella classe `B`.



NB: Nell'overriding bisogna prestare attenzione alla segnatura dei metodi.

Dato il metodo virtuale di una classe `B`:

```
virtual T m(T1,T2,...,Tn);
```

allora l'overriding di `m()` in una classe `D` derivata da `B` deve mantenere la stessa segnatura, incluso il tipo di ritorno.

NB: Se effettuo overriding di un metodo virtuale `m()` in una classe `D` derivata da `B` \Rightarrow in `D` vengono nascosti tutti gli eventuali overloading di `m()` in `B`.

Es:

```
class B{
public:
    virtual int f(){ cout<<"B::f()"; return 1; }
    virtual void f(string s){ cout<<"B::f(string)"; }
    virtual void g(){ cout<<"B::g()"; }
};
```

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

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

```
class D3: public B{
public:
    void f(){ cout<<"D3::f()"; } //ERRORE, non è possibile modificare il tipo di ritorno
};
```

```
class D4: public B{
public:
    int f(int){ cout<<"D4::f()"; return 4; } //La lista degli argomenti è stata modificata,
                                                //è però una ridefinizione, non un overriding
};
```

```
int main(){
    string s = "ciao";
    D1 d1; D2 d2; D3 d3; D4 d4;
    int x = d1.f(); //OK, stampa B::f() metodo virtuale
    d1.f(s); //OK, stampa B::f(string)
    x = d2.f(); //OK, stampa D2::f()
    d2.f(s); //ERRORE, overload di f nascosto a D2
    x = d4.f(1); //OK, stampa D4::f()
    x = d4.f(); //ERRORE, f() di B ridefinita in B è nascosta a D4
    d4.f(s); //ERRORE, "" "" ""
    B& br = d4;
    br.f(1); //ERRORE, f(int) non è disponibile in B
    br.f(); //OK, stampa B::f()
    br.f(s); //OK, stampa B::f(string)
}
```

L'unica eccezione a questa regola è nel caso in cui il tipo di ritorno sia tipo puntatore o riferimento ad una classe:

Dato il metodo virtuale dove `X` è un tipo di classe:

```
virtual X* m(T1,T2,...,Tn);
```

allora se `Y` è sottoclasse di `X`, è permesso che l'overriding di `m()` possa cambiare il tipo di ritorno in `Y*` (mentre la lista dei parametri deve rimanere la stessa)

Ciò è permesso perché l'overriding `virtual Y* m(T1,T2,...,Tn);` ritorna un puntatore a `Y` sottoclasse di `X`, e quindi questo puntatore può essere convertito implicitamente in un puntatore ad `X`. (analogo per i riferimenti)

Es:

```
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; } //OK, overriding di m() ha stessa segnatura
};
```

Ritorna un oggetto `Y` per poi sfruttare la conversione implicita da `Y*` \Rightarrow `X*`

```
class D: public B{
    Z z;
public:
    virtual Z* m(){ return &z; } //OK, overriding modifica tipo ritorno X*  $\Rightarrow$  Z*
};
```

Ritorna un puntatore ad un oggetto di `Z` e non necessita di nessuna conversione implicita grazie al cambiamento del tipo di ritorno

```
int main(){
    C c; D d;
    Y* py=c.m(); //ERRORE, non è possibile accedere al campo dati di tipo Y dell'oggetto "c" dato che l'overriding di m() ritorna un puntatore ad X
    X* px=c.m(); //OK, è possibile accedere al campo dati di tipo X di "c" in quanto ritorna un tipo Y che viene convertito in X
    Z* pz=d.m(); //OK, è possibile accedere al campo dati di tipo Z di "d" dato che l'overriding di m() ritorna un puntatore a Z
}
```

NB: Bisogna prestare attenzione all'overriding di metodi virtuali che hanno parametri che prevedono valori di default.

In questo caso il linguaggio non prevede che la segnatura del metodo debba per forza ripetere i valori di default, che possono essere omissi.

⇒ Si tratta sempre di overriding di un metodo anche se si omettono i valori di default presenti nel metodo di cui fa l'override
(non si tratta quindi di un nuovo metodo della classe derivata)

Es:

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

```
class D: public B{
public:
    virtual void m(int x){ cout<<"D::m"; } //overriding del metodo virtuale B::m
    virtual void m(){ cout<<"D::m()"; } //nuovo metodo in D e non overriding di B::m
};
```

```
int main(){
    B* p = new D;
    D* q = new D;
    p->m(2); //stampa D::m perchè il tipo dinamico di p è D*
    p->m(); //stampa D::m e non D::m() in quanto il tipo
           //dinamico di p è D*
    q->m(); //stampa D::m() e non D::m in quanto il tipo
           //statico = tipo dinamico di p è D*
}
```

NB: E' possibile bloccare il late binding di un metodo virtuale tramite l'uso dell'operatore di scoping:

Es:

```
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 ha tipo statico C* e tipo dinamico D*
    p->m(); //dynamic binding, dato che il tipo dinamico è D* si provoca l'invocazione dell'overriding D::m()
    p->B::m(); //static binding, lo scoping blocca il dynamic binding, quindi stampa B::m()
    p->C::m(); //static binding, "" "", quindi stampa C::m()
}
```

Una **Chiamata Polimorfa** di un metodo (cioè l'invocazione di un metodo virtuale tramite un puntatore polimorfo), provoca effetti diversi a seconda del tipo dinamico del puntatore. Ciò promuove l'estensibilità del software, infatti i programmi che sfruttano il polimorfismo hanno un comportamento indipendente dal tipo statico di puntatori e riferimenti che i metodi virtuali invocano.

Esempio: uno screen manager è un modulo software che si occupa di gestire la visualizzazione di vari "oggetti" su uno schermo. Dovrà essere in grado di gestire una varietà di oggetti di tipi diversi, e con tipi che saranno aggiunti al sistema globale anche dopo la sua scrittura. Lo screen manager non si preoccupa del particolare oggetto da visualizzare, semplicemente ordina all'oggetto in questione di visualizzarsi in modo autonomo, qualunque esso sia:

- Esso userà puntatori e riferimenti ad una classe base "Shape" per gestire tutti gli oggetti da visualizzare.
- Shape conterrà un metodo virtuale draw() e per visualizzare l'oggetto lo screen manager invocherà semplicemente p->draw(); con p puntatore alla classe base Shape;
- draw() dovrà quindi essere ridefinito in tutte le sottoclassi di Shape, ogni particolare sottoclasse saprà dunque come visualizzare un proprio oggetto.

6.5.2 Distruttori Virtuali

Data B classe base e D classe derivata da B:

```
D* pd = new D;
B* pb = pd; //qui TD(pb)=D*
delete pb;
```



Con delete pb; viene invocato il distruttore della classe base B su un oggetto della classe derivata D.

Per evitare questo si deve dichiarare virtuale il distruttore della classe base B, in questo modo tutti i distruttori delle classi derivate da B diventano automaticamente virtuali in modo che se viene applicata la delete ad un puntatore della classe base B il cui tipo dinamico è D* (con D sottoclasse di B) ⇒ allora viene invocato il distruttore di D.

NB: In una classe che contiene metodi virtuali è buona norma includere un distruttore virtuale, anche se non è strettamente necessario.

Spesso si dichiara il **distruttore virtuale con corpo vuoto**, in questo modo il **distruttore standard verrà reso virtuale**. (non vale per i costruttori)

Es:

```
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 ~C(){ delete[] q; cout<<"~C()"; } //distruttore virtuale
};
```

```
int main(){
    C* q = new C(4, 2, 18);
    B* p = q; //puntatore polimorfo con TS(p)=B* e TD(p)=C*
    delete p; //dato che TD(p)=C* invoca il distruttore di ~C
} //stampa ~C|~B (prima distruttore per la classe
    derivata C e poi per la classe base B
```

```
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"; }
};
```

6.5.3 Metodi Virtuali Puri

In alcune circostanze, una classe base B viene progettata solo per essere usata successivamente per definire delle classi derivate da B e non per creare oggetti di B veri e propri. In questo caso tale classe base B funziona da "interfaccia comune" per le sue classi derivate, cioè B si limita a specificare la lista delle operazioni basilari che caratterizzano tutti i sottotipi di B.

Un esempio di ciò è la classe base "interfaccia" Shape, considerata nell'esempio dello screen manager, che specifica l'esistenza di un metodo virtuale draw() che le classi derivate da Shape implementano con diversi modi specifici la visualizzazione dei loro oggetti.

Altro esempio è la classe base "interfaccia" UnitadiOutput, con le sue classi derivate Video1, Stampante1, Video2, Stampante2 etc. In questo caso sarebbe inutile scrivere il codice dei metodi virtuali della classe base UnitadiOutput che permettono di visualizzare un suo oggetto, dato che non abbiamo sufficienti dettagli rappresentativi che ci indichino cosa visualizzare a schermo, cosa che invece sarà definita nelle sue classi derivate.

Il C++ permette quindi di dichiarare un metodo virtuale senza definirne il corpo attraverso il marcatore "=0" alla fine della sua dichiarazione.

```
class B{
    virtual void G()=0;
    ...
};
```

Il metodo G() è dichiarato **metodo virtuale puro**.

La classe base B è detta **classe astratta** se contiene o eredita, senza definirlo, almeno un metodo virtuale puro.

La sottoclasse D di una classe base astratta B è detta **classe concreta** se D implementa tutti i metodi virtuali puri di B.

NB: Se si tenta di istanziare oggetti di una classe base astratta, il compilatore segnalerà errore in quanto **gli oggetti di una classe base astratta B possono essere creati soltanto come sottooggetti di oggetti appartenenti ad una sottoclasse concreta di B**. (una classe astratta può comunque contenere costruttori)

Es:

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

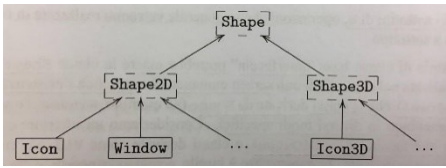
```
class C : public B{ }; //sottoclasse astratta
```

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

```
int main(){
    C c; //ERRORE, C non ha tipo in quanto vuota
    D d; //OK, dichiaro d di tipo D classe concreta
    B* p; //OK, puntatore a classe astratta
    p = &d; //p puntatore polimorfo con TD(p)=D*
    p->f(); //OK, stampa D::f() dato che TD(p)=D*
}
```

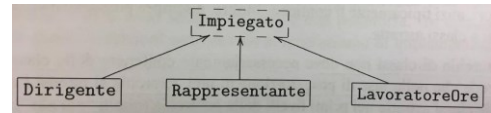
Una gerarchia di classi non deve per forza contenere classi astratte, anche se spesso succede che nella fase di progettazione di una gerarchia vi sia l'opportunità di definire classi astratte nei primi livelli della gerarchia.

Es:



nello screen manager, la gerarchia potrebbe partire dalla classe astratta Shape che potrebbe avere due sottoclassi di Shape ancora astratte Shape2D e Shape3D, mentre dal terzo livello si potrebbe cominciare a definire delle sottoclassi concrete.

Es: vogliamo definire una classe astratta Impiegato da cui derivano tre classi concrete Dirigente, Rappresentante e LavoratoreOre.



```

class Impiegato{ //Classe Base Astratta
private:
    string nome;
public:
    Impiegato(string s): nome(s){ }
    string getNome() const{ return nome; }
    virtual double stipendio() const = 0; //metodo virtuale puro
    virtual void print() const{ cout<<nome; } //metodo virtuale
};
    
```

```

class Dirigente : public Impiegato{
private:
    double fissoMensile; //stipendio fisso mensile
public:
    Dirigente(string s, double d=0): Impiegato(s), fissoMensile(d){ } //costruttore
    void setFissoMensile(double d){ //metodo non costante
        fissoMensile= d > 0 ? d : 0;
    }
    virtual double stipendio() const{ //implementazione metodo virtuale puro
        return fissoMensile;
    }
    virtual void print() const{ //overriding di Impiegato::print()
        cout<<"Il dirigente";
        Impiegato::print(); //invocazione statica di Impiegato::print()
    }
};
    
```

```

class Lavoratore : public Impiegato{
private:
    double pagaOraria;
    double oreLavorate; //ore lavorate nel mese
public:
    Lavoratore( string s, double d=0, double e=0): Impiegato(s),
        pagaOraria(d), oreLavorate(e){ }
    void setPaga(double d){ pagaOraria= d > 0 ? d : 0; }
    void setOre(double d){ oreLavorate = d >= 0 && d <= 250 ? d : 0; }
    virtual double stipendio() const{ //implementazione metodo virtuale puro
        if(oreLavorate <= 160) return pagaOraria*oreLavorate; //se non ha straordinari
        else return 160*pagaOraria+(oreLavorate - 160)*2*pagaOraria;
    }
    virtual void print() const{ //overriding di Impiegato::print()
        cout<<"Il lavoratore"; Impiegato::print();
    }
};
    
```

```

class Rappresentante : public Impiegato{
private:
    double baseMensile; //stipendio base fisso
    double commissione; //commissione per pezzo venduto
    int tot; //pezzi venduti in un mese
public:
    Rappresentante(string s, double d=0, double e=0, int x=0): Impiegato(s),
        baseMensile(d), commissione(e), tot(x){ }
    void setBase(double d){ baseMensile = d > 0 ? d : 0; }
    void setCommissione(double d){ commissione = d > 0 ? d : 0; }
    void setVenduti(int x){ tot = x > 0 ? x : 0; }
    virtual double stipendio() const{ //implementazione metodo virtuale puro
        return baseMensile+ commissione*tot;
    }
    virtual void print() const{ //overriding di Impiegato::print()
        cout<<"Il rappresentante"; Impiegato::print()
    }
};
    
```

```

void StampaStipendio(Impiegato* p){
    p->print(); //chiamata polimorfa
    cout<<"in questo ha guadagnato :."
    cout<< p->stipendio() << "€" ; //chiamata polimorfa
};
    
```



```

int main(){
    Dirigente d("Paperino", 4000);
    Rappresentante r("Topolino", 1000, 3, 250);
    LavoratoreOre l("Pluto", 15, 170);
    stampaStipendio(&d); //Il dirigente Paperino mese ha guadagnato 4000€
    stampaStipendio(&r); //L'impiegato Topolino ha guadagnato 1750€
    stampaStipendio(&l); //Il lavoratore a ore Pluto ha guadagnato 2700€
}
    
```

6.6 Identificazione di tipi a run-time

Il C++ permette di determinare il tipo dinamico di un puntatore o riferimento a tempo di esecuzione tramite gli operatori **typeid** e **dynamic_cast**.

6.6.1 typeid

L'operatore **typeid**, definito nella libreria `typeinfo`, permette di determinare il tipo di una qualsiasi espressione a tempo di esecuzione.

Se l'espressione è un riferimento o un puntatore polimorfo dereferenziato, allora **typeid** ritornerà il tipo dinamico di questa espressione.

```
#include<typeinfo>
#include<iostream>
using std::cout;
using ::endl;

int main(){
    int i = 5;
    cout<< typeid(i).name() <<endl;      //Stampa: i(nt)
    cout<< typeid(3.14).name() <<endl;   //Stampa d(ouble)
    if( typeid(i) == typeid(int) ) cout<<"Yes";
}
```

L'operatore **typeid** ha come argomento un'espressione o un tipo qualsiasi, e ritorna un oggetto della classe `type_info`.

La definizione della classe `type_info` è nel file header `typeinfo` ed ogni implementazione include almeno i seguenti modi:

```
class type_info{ //ha rappresentazione dipendente dall'implementazione
private:
    type_info();
    type_info(const type_info&);
    type_info& operator=(const type_info&);
public:
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    const char* name() const;
};
```

-Non è possibile dichiarare, modificare o assegnare oggetti di tipo `type_info`, dato che l'unico costruttore è privato, come lo sono la copia e l'assegnazione.

NB: è solamente possibile confrontare oggetti `type_info` con operatori di uguaglianza e disuguaglianza ed estrarne il loro nome tramite `name()`.

Comportamento:

-Dato ref riferimento ad una classe che contiene almeno un metodo virtuale

⇒ allora **typeid(ref)** restituisce un oggetto di `type_info` che rappresenta il tipo dinamico di ref.

-Dato `*punt` puntatore dereferenziato che punta ad una classe che contiene almeno un metodo virtuale ⇒

⇒ allora **typeid(*punt)** restituisce un oggetto `type_info` che rappresenta T dove T* è il tipo dinamico del puntatore.

- NB:**
- Un tipo T è polimorfo se è un tipo classe che include almeno un metodo virtuale;
 - Una classe polimorfa C è una classe che contiene solo il distruttore virtuale;
 - Ogni sottotipo di un tipo polimorfo è a sua volta un tipo polimorfo ⇒ quindi **ogni classe derivata da una classe polimorfa è polimorfa**.

- Inoltre:
- a. Se una classe non è polimorfa, cioè non contiene metodi virtuali ⇒ allora **typeid** restituisce il tipo statico del rif/punt dereferenziato;
 - b. Se uso **typeid** su un puntatore non dereferenziato ⇒ allora **typeid** restituisce sempre il tipo statico del puntatore.
 - c. **typeid ignora sempre l'attributo const**: ad es l'espressione `typeid(T)==typeid(const T)` è valutato sempre true.

6.6.1 dynamic_cast

L'operatore di conversione esplicita **dynamic_cast** permette di convertire puntatori e riferimenti ad una classe base B polimorfa in puntatori e riferimenti ad una classe D derivata da B: **-B* ⇒ D***

-B& ⇒ D&

Al contrario degli altri operatori di conversione del C++, il **dynamic_cast** è eseguito a run-time dato che il successo della conversione dipende dal tipo dinamico del puntatore/riferimento che vogliamo convertire.

Per usare il **dynamic_cast** la classe B deve essere polimorfa, altrimenti ci sarà errore.

■ Dato p puntatore ad una classe polimorfa B che punta a qualche suo oggetto obj e D sottoclasse di B:

```
dynamic_cast<D*>(p);
```



permette di convertire di p al "tipo target" D* solo nel caso in cui il tipo dinamico del puntatore p sia quello giusto.

Dato `TD(p) = E*` :

- Se E è sottotipo di D ⇒ OK, la **conversione andrà a buon fine** ⇒ **dynamic_cast** ritornerà un puntatore di tipo D* all'oggetto obj. (effettua conversione dato che il tipo dinamico E* di p è compatibile con il tipo target D*, in quanto E* è sottotipo di D*)
- Se E non è sottotipo di D ⇒ **ERRORE**, la **conversione fallisce** e il **dynamic_cast** ritornerà un puntatore nullo. (in questo caso il tipo dinamico del puntatore non è compatibile con il tipo target)

```
class Padre {.....};
class Figlio : public Padre {.....};
class Nipote : public Figlio {.....};
```



```
main () {
    Padre *p, *pObj = new Padre;
    Figlio *f, *fObj = new Figlio;
    Nipote *n, *nObj = new Nipote;

    p = nObj;    // OK ⇒ un puntatore ad una classe derivata può essere assegnato ad uno di classe base
    n = p;       // ERRORE ⇒ non sa se l'assegnamento è valido (lo si sa solo run-time)
    n = dynamic_cast<Nipote*>(p);    // qui si chiede di fare il controllo run-time per validare la conversione
                                     (valida in questo caso, dato che il TD(p)=Nipote)

    if (n) cout << "conversione eseguita con successo";

    p = fObj;    // OK ⇒ puntatore ad una classe derivata può essere assegnato ad uno di classe base
    n = dynamic_cast<Nipote*>(p);    // qui si chiede di fare il controllo run-time per validare la conversione
                                     (non valida in questo caso, dato che TD(p)=Figlio)

    if (!n) cout << "conversione fallita";
}
```

Safe Downcasting: quando si converte “dall’alto verso il basso” nella gerarchia tramite `dynamic_cast`
 ⇒ cioè da classe base a classe derivata: $B^* \Rightarrow D^*$ oppure $B\& \Rightarrow D\&$ tramite

```
dynamic_cast<D*>(p); //Downcast
```

Upcasting: quando si converte “dal basso verso l’alto” nella gerarchia tramite un assegnazione
 ⇒ cioè da classe derivata a classe base: $D^* \Rightarrow B^*$ oppure $D\& \Rightarrow B\&$ per esempio con:

```
D d;
B* b = d; //Upcast
```

Nel caso dei riferimenti, se il `dynamic_cast` di un riferimento fallisce, viene automaticamente lanciata un’eccezione di tipo `bad_cast`:

```
class X{ public: virtual ~X() };
class B{ public: virtual ~B() };
class D: public B{};
```



```
#include<typeinfo>
#include<iostream>
using namespace std;

int main(){
    D d;
    B& b = d;//Upcasting
    try{ X& xr=dynamic_cast<X&>(b); }
    catch(bad_cast e){ cout<<"Conversione fallita"; }
}
```

NB: Il safe downcasting va usato solo in caso di necessità, in genere si fa downcasting di un puntatore ad una classe base B quando abbiamo bisogno di ottenere dei membri di una classe derivata da B che non sono ereditati da B.

Es. d’uso di safe downcasting:

```
class Impiegato{    //classe astratta
public:
    virtual double stipendioBase() const=0;
};
```

```
class Programmatore: public Impiegato{
public:
    virtual double stipendioBase() const{ return 1500; }
    virtual double bonus() const{ return 300; } //metodo specifico di Programmatore
};
```

```
class Manager: public Impiegato{
public:
    virtual double stipendioBase() const{ return 2000; }
};
```

```
class SoftwareHouse{
public:
    static double stipendio(const Impiegato& p){
        const Programmatore* q = dynamic_cast<const Programmatore*>(&p);
        if(q) return q->stipendioBase() + q->bonus();    //se il cast ha successo, cioè se TD(&p) è sottotipo di Programmatore* allora ritorna stipendio+bonus
        else return p.stipendioBase();    //altrimenti se q==0, cioè se TD(&p) non è sottotipo di Programmatore*, basta lo stipendio base
    }
};
```

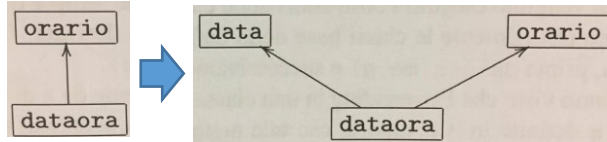
```
int main(){
    Manager m;
    Programmatore p;
    cout<<"Il Manager guadagna: ";
    cout<<SoftwareHouse::stipendio(m)<<" €"<<endl;

    cout<<"Il Programmatore guadagna: ";
    cout<<SoftwareHouse::stipendio(p)<<" €"<<endl;
}
```

6.7 Ereditarietà Multipla

Una classe può essere derivata da più di una classe base, in questo caso ogni oggetto della classe derivata contiene un sottooggetto per ognuna delle classi base da cui deriva.

Es: invece di definire `dataora` come derivata della classe base `orario` aggiungendo dei campi dati che gestiscono la data, possiamo invece definire una seconda classe base "data" i cui oggetti rappresentano una data per poi definire la classe `dataora` come derivata sia dalla classe base `orario` che dalla classe base `data`.



```
class data{
public:
    data(int=1, int=1, int=0);
    int Giorno() const{ return giorno; }
    int Mese() const{ return mese; }
    int Anno() const{ return anno; }
protected:
    int giorno, mese, anno;
    void AvanzaUnGiorno();
private:
    int GiorniDelMese() const;
    bool Bisestile() const;
};
```

```
class dataora: public data, public orario{
public:
    dataora(){} //costruttore default => richiama implicitamente i costruttori di default per i sottooggetti delle classi base
    dataora(int a, int me, int g, int o, int m, int s): data(a,m,g), orario(o,m,s) {}
    //costruttore a sei parametri richiama esplicitamente i costruttori delle classi base
    dataora operator+(const orario&) const;
    bool operator==(const dataora&) const;
    ...
}
```

NB: la tipologia di derivazione (public, private, protected) per le classi che derivano da più classi base può essere diversa per ogni classe base.

6.7.1 Ambiguità:

Abbiamo visto che l'overriding in una classe D derivata direttamente da B di un metodo di nome m definito in B, comporta che tale metodo m di B, e tutti gli eventuali metodi sovraccaricati di nome B, non siano visibili in D a meno che non si usi l'operatore di scoping B::

NB: Sia D classe che deriva da B e C in cui sono presenti **due metodi distinti con lo stesso nome m()**, cioè si hanno **B::m()** e **C::m()**

⇒ **Se su D si prova ad invocare m() si causa un errore di compilazione** in quanto il compilatore non è in grado di decidere quali metodi invocare a causa dell'ambiguità **del nome di m()**. Per risolvere l'ambiguità si deve usare l'operatore di scoping.

Es: se sia sulla classe `orario` che sulla classe `data` c'è un metodo `Stampa()` che stampa un'orario ed una data, allora a causa dell'ambiguità dell'invocazione di `Stampa()`, il compilatore non è in grado di decidere quale dei due metodi utilizzare.
(l'errore verrà notificato solo quando si cercherà di utilizzare il metodo `Stampa()`, non durante la compilazione).

```
void orario::Stampa() const{
    cout<< Ore() <<":"<<Minuti()<<":"<<Secondi();
}
void data::Stampa() const{
    cout<< Giorno() <<"/"<< Mese()<<"/"<< Anno();
}
```

```
dataora d;
d.Stampa(); //ERRORE, la classe dataora eredita due funzioni diverse con lo stesso nome Stampa()
```

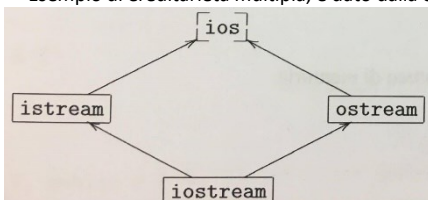
Per risolvere l'ambiguità, sarà necessario utilizzare l'operatore di scoping:

```
dataora d;
d.data::Stampa(); //OK
d.orario::Stampa(); //OK
```

Oppure ridefinire il metodo `Stampa()` nella classe `dataora` in modo da nascondere entrambi i metodi `Stampa()` delle classi base ed escludere qualsiasi ambiguità:

```
void dataora::Stampa() const{
    data::Stampa();
    orario::Stampa();
}
```

Esempio di ereditarietà multipla, è dato dalla **classe iostream** della gerarchia di classi di input/output della libreria standard.

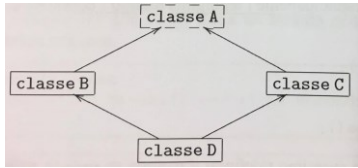


La classe `ios` è base sia per la classe `ostream` che per la classe `istream`.
La classe `iostream` deriva direttamente sia da `ostream` che da `istream`.

Ciò consente agli oggetti di tipo `iostream` di fornire le funzionalità sia di `istream` che di `ostream`.

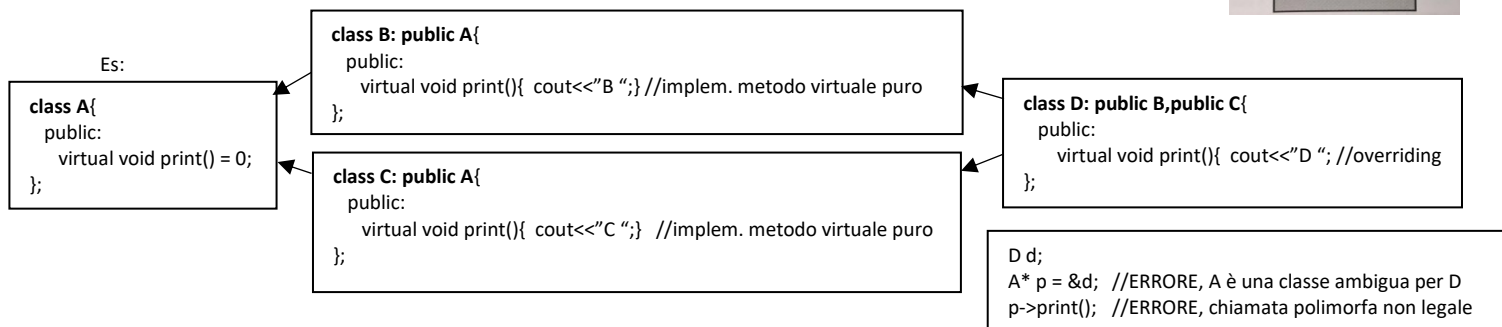
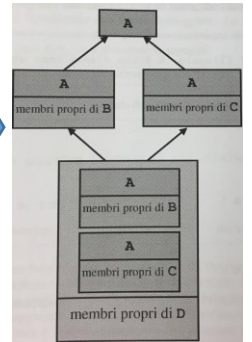
6.7.1 Derivazione Virtuale

Una gerarchia di classi può essere molto complessa, può capitare che una classe derivi da due classi base le quali a loro volta derivano, direttamente o indirettamente, da una stessa classe. In questo caso si parla di **ereditarietà a diamante** (come nel caso di iostream).

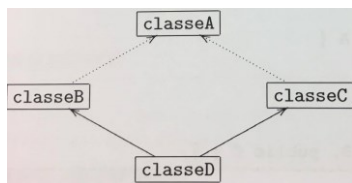


In questo caso si ha che ogni oggetto della classe D, contiene due sottooggetti della classe A. Ciò comporta due problemi:

- **Produce ambiguità** che impedisce di accedere a tali sottooggetti ed ai metodi che operano su di essi;
- **Produce spreco di memoria.**



La soluzione sarebbe quella di avere un unico sottooggetto di classe A in ogni oggetto della classe D:



Ciò si può realizzare utilizzando la **derivazione virtuale** (freccie tratteggiate):

```

class A{ ..... }; //A è classe base virtuale per le classi B e C
class B: virtual public A{ .... };
class C: virtual public A{ ... };
class D: public B, public C{ ... };
  
```

NB: Data A classe virtuale, per ogni classe D derivata da A e definita tramite derivazione multipla ⇒ Allora ci sarà soltanto un sottooggetto di A in ogni oggetto della classe D.

Nella pratica, la **derivazione virtuale** è implementata tramite puntatori:

- Gli oggetti delle classi B e C derivate virtualmente da A, oltre al sottooggetto "ordinario" della classe A, contengono un puntatore ad un oggetto della classe A;
- Gli oggetti della classe D conterranno invece un puntatore ad un unico sottooggetto di tipo A.

Esempio senza derivazione virtuale:

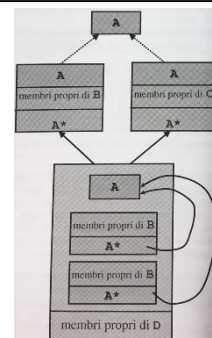
class A{ int a[5]; }; //20 byte	➡	cout<< sizeof(A) << endl; //stampa 20
class B: public A{ int y[3]; }; //12 byte		cout<< sizeof(B) << endl; //stampa 12+20=32
class C: public A{ int z[3]; }; //12 byte		cout<< sizeof(C) << endl; //stampa 12+20=32
class D: public B, public C{ int w[4]; }; //16 byte		cout<< sizeof(D) << endl; //stampa 80=16+32+32

Esempio con derivazione virtuale, qui la memoria utilizzata è inferiore:

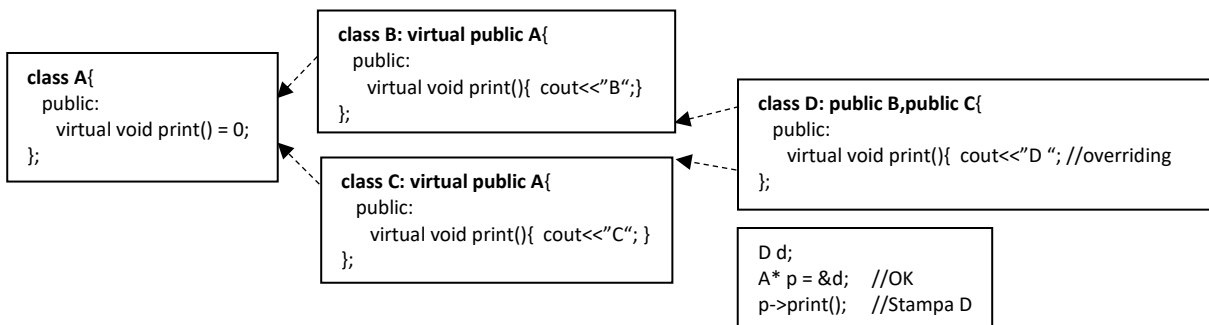
class A{ int a[5]; }; //20 byte	➡	cout<< sizeof(A) << endl; //stampa 20
class B: virtual public A{ int y[3]; }; //12 byte + 1 puntatore		cout<< sizeof(B) << endl; //stampa 12+4+20=36
class C: virtual public A{ int z[3]; }; //12 byte + 1 puntatore		cout<< sizeof(C) << endl; //stampa 12+4+20=36
class D: public B, public C{ int w[4]; }; //16 byte		cout<< sizeof(D) << endl; //stampa 68=16+(12+4)+(12+4)+20

In questo caso, la sizeof(D) è data dalla somma delle dimensioni:

- dell'array proprio di D w(16 byte);
- dell'array proprio di B y(12 byte);
- dell'array proprio di C z(12 byte);
- dei due puntatori che implementano la derivazione virtuale contenuti in B e C (4 byte+4byte);
- dell'unico array a(20 byte) dell'unico sottooggetto di A condiviso in D.



Es: come quello di prima ma con la derivazione virtuale

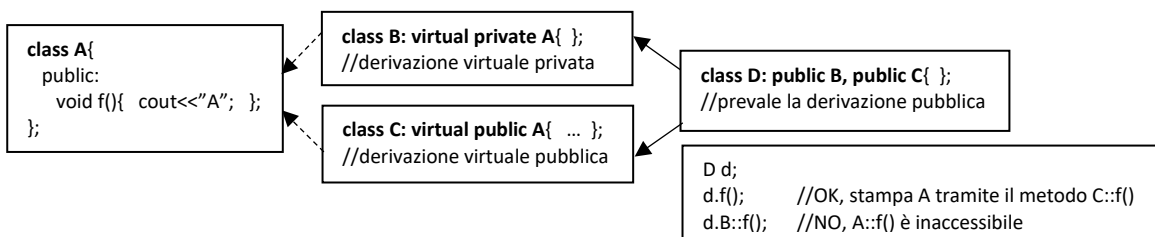


Anche per la derivazione virtuale multipla, possiamo avere derivazione privata, pubblica o protetta.

In una **classe derivata si può quindi avere la stessa classe base virtuale indiretta ma con diverse regole d'accesso.**

In questo caso vale la regola:

- la **derivazione protetta prevale sulla privata**;
- la derivazione pubblica prevale sulla protetta.



Sappiamo che in un costruttore di una classe derivata richiama preliminarmente (implicitamente o esplicitamente) soltanto i costruttori delle sue superclassi dirette.

NB: In presenza di classi basi virtuali, il **costruttore di una classe derivata, oltre che richiamare preliminarmente i costruttori delle sue superclassi dirette, richiama inoltre preliminarmente anche i costruttori delle classi virtuali presenti nella sua gerarchia di derivazione.** (implicitamente o esplicitamente tramite un'invocazione inclusa nella lista di inizializzazione).

Ciò è necessario in quanto il **sottoggetto di una classe base virtuale A è condiviso**, e quindi non avrebbe senso costruirlo più di una volta.
⇒ Perciò **il sottoggetto della classe base virtuale A va costruito prima che avvenga la sua condivisione.**

Comportamento costruttore di una classe derivata D con ereditarietà multipla e classi virtuali:

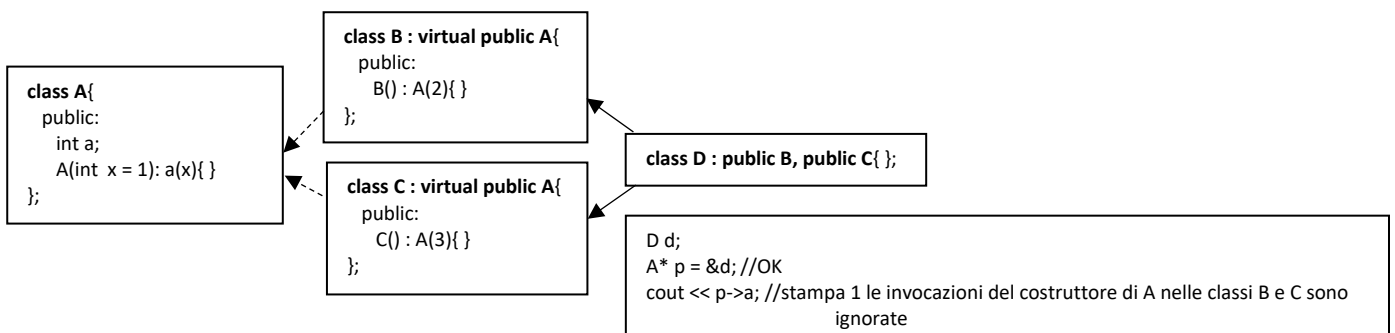
1. Vengono richiamati, una sola volta, i costruttori delle classi virtuali che si trovano nella gerarchia di derivazione di D.
⇒ Se vi è più di una classe virtuale in D ⇒ 1. La ricerca delle classi virtuali procede esaminando per prime le superclassi dirette di D in ordine di dichiarazione di ereditarietà (da sx verso dx nella gerarchia);
2. Per ognuna di queste sottogerarchie si cercano ricorsivamente le classi base virtuali.
2. Dopo che sono stati invocati i costruttori delle classi virtuali nella gerarchia di derivazione di D
⇒ vengono richiamati i costruttori delle superclassi dirette non virtuali di D (NB: questi costruttori escludono di richiamare eventuali costruttori di classi virtuali già richiamati al passo 1);
3. Viene eseguito il costruttore proprio di D ⇒ 1. Campi propri;
2. Corpo del costruttore di D.

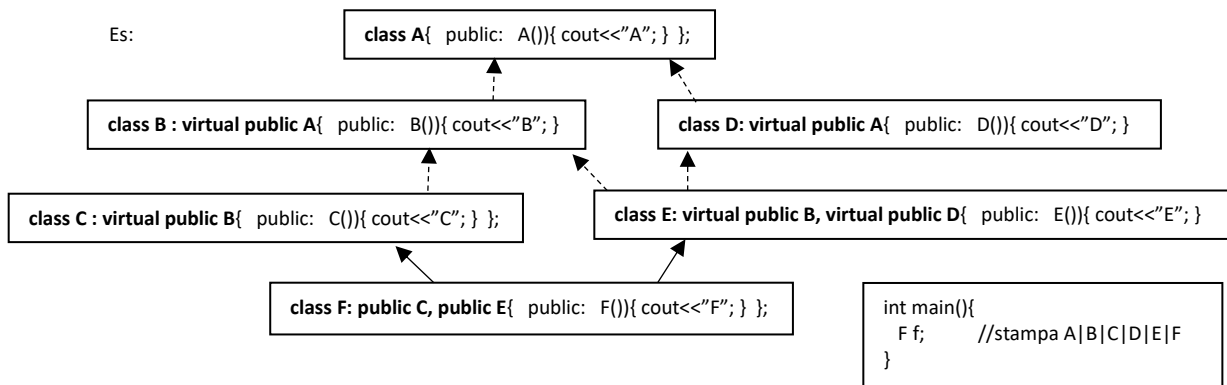
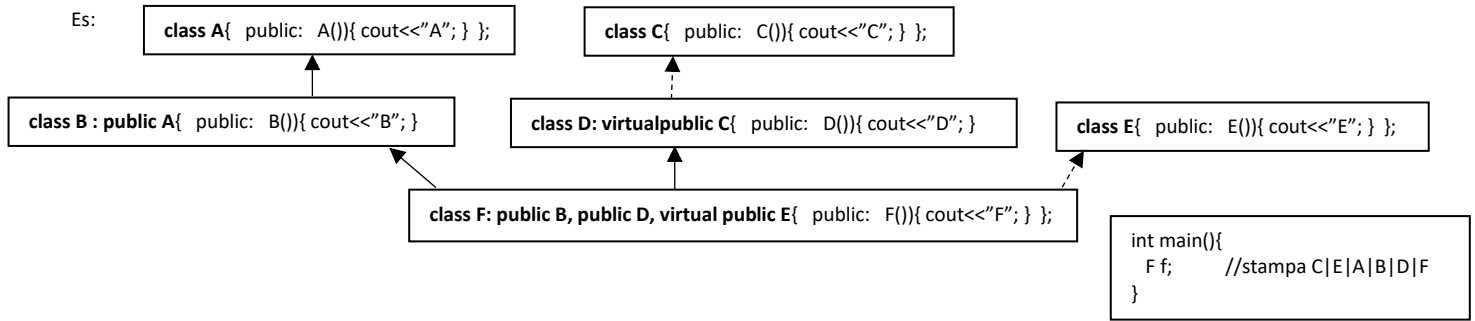
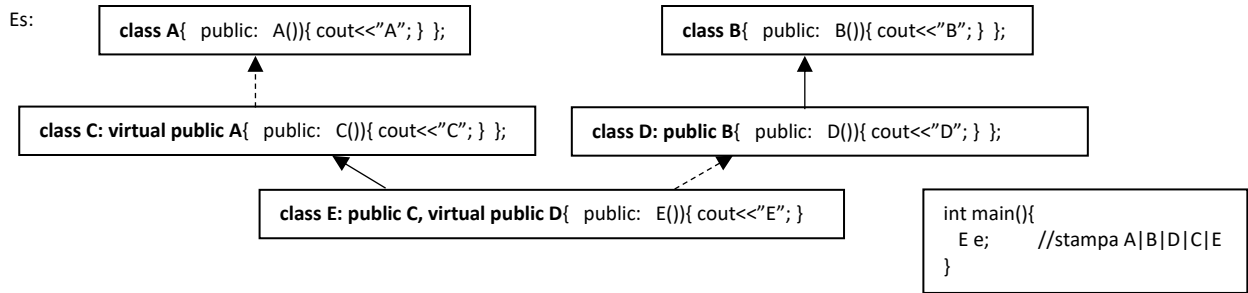
Nel caso "a diamante" precedente, il compito di costruire il sottoggetto della classe base virtuale A viene lasciato alla classe dell'oggetto principale che stiamo definendo, nel nostro caso al costruttore di D, perciò:

NB: Un'eventuale chiamata esplicita al costruttore della classe virtuale A deve essere messa nella lista di inizializzazione di D.

Di conseguenza **se nelle liste di inizializzazione delle classi B e C compare un'invocazione del costruttore di A**
⇒ nella costruzione di un oggetto di D tale chiamata viene ignorata.

Es:





Es: Definire una classe "cilicono" i cui oggetti rappresentano una figura geometrica solida composta da un cono sovrapposto ad un cilindro con la stessa base. E' presente una gerarchia di classi a diamante con una classe base virtuale ed astratta.

```
class solidoConCerchio{
protected:
    double raggio;
    double circonferenza() const{ return (2*M_PI*raggio); }
    double areaCerchio() const{ return (M_PI*raggio*raggio); }
public:
    solidoConCerchio(double r): raggio(r){ }
    virtual double area() const = 0; //virtuale puro
    virtual double volume() const = 0; //virtuale puro
};
```

```
class cilindro: virtual public solidoConCerchio{ //derivazione virtuale
protected:
    double altezza;
    double areaLaterale() const { return circonferenza()*altezza; }
public:
    cilindro(double a, double h): solidoConCerchio(a), altezza(h){ }
    virtual double area() const{ return (2 * areaCerchio() + areaLaterale()); }
    virtual double volume() const{ return (areaCerchio() * altezza); }
};
```

```
class cono: virtual public solidoConCerchio{ //derivazione virtuale
protected:
    double altezza;
    double areaLaterale() const{ double apotema = sqrt(raggio * raggio + altezza * altezza);
                                return (M_PI * raggio * apotema); }
public:
    cono(double a, double h): solidoConCerchio(a), altezza(h){ }
    virtual double area() const { return (areaCerchio() + areaLaterale()); }
    virtual double volume() const{ return (areaCerchio() * altezza / 3; }
};
```

```
class cilicono: public cilindro, public cono{ //derivazione multipla in modo da avere solo un
public:
    sottooggetto di "solidoConCerchio"
    cilicono(double r, double h1, double h2): solidoConCerchio(r), cilindro(r, h1), cono(r, h2) { }
    virtual double area() const{ return (cilindro::areaLaterale() + cono::areaLaterale() + areaCerchio()); }
    virtual double volume() const{ return (cilindro::volume() + cono::volume()); }
};
```

```
int main(){
    cilindro cil(1,2);
    cono co(1,2);
    cilicono clc(1,2,2);
    cout<< " Area Cilindro: "<< p->area() << endl;
    cout<<" Volume Cilindro: "<< p->volume() << endl;
    p = &co;
    cout<< " Area cono: "<< p->area() << endl;
    cout<< " Volume cono: "<< p->volume() << endl;
    p = &clc;
    cout<< " Area cilicono: "<< p->area() << endl;
    cout<< " Volume cilicono: "<< p->volume() << endl;
}
```

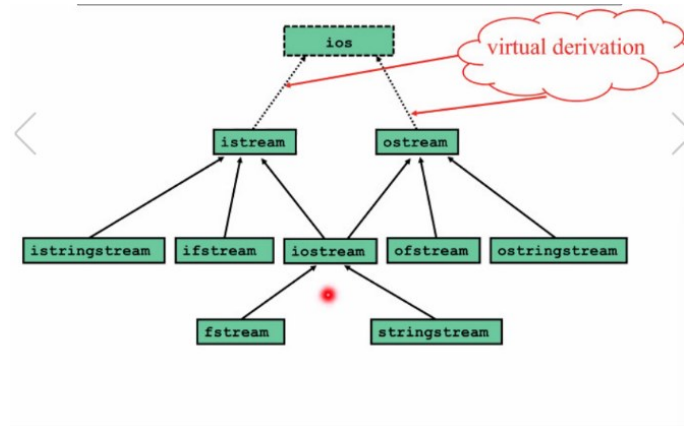
NB: Notare che se il costruttore della classe cilicono non invocasse esplicitamente il costruttore di solidoConCerchio con "solidoConCerchio(r)" nella lista di inizializzazione del suo costruttore, allora tale definizione non compilerebbe dato che il costruttore di default di solidoConCerchio non è disponibile.

CAPITOLO 8 – “Gerarchia di Classi per l’I/O”

Gli oggetti delle varie classi di I/O vengono detti **stream** i quali:

- Sono formati da sequenze non limitate di celle ognuna delle quali contenente un byte;
- La posizione di una cella di uno stream è un intero che parte da 0 (come negli array);
- Ogni stream ha associato un buffer attraverso il quale passano effettivamente le operazioni di I/O;

NB: Per utilizzare le superclassi di iostream bisogna includere il file header <iostream>, mentre il file header <fstream> contiene le dichiarazioni per le classi fstream, ifstream, ofstream. Bisogna includere le librerie “fstream” e “sstream” per usare rispettivamente metodi di utility e apertura file (out, in, trunc, ate, etc.) oppure per la classe *stringstream*.



La classe base astratta virtuale degli input è *ios_base*, da cui deriva *ios* (si può vedere che è derivata da *std*), la quale possiede alcuni bit particolari, indicati con 0 od 1, dal più significativo (bad, errore fatale/fisico, per cui non è possibile continuare, poi fail, errore senza perdita di dati od eof, fine file) al meno.

La stessa classe *istream* è virtuale pubblica e considera tutti gli operatori; ogni operatore di input ignora le singole spaziature/spazi. Se l’input fallisce non viene effettuato alcun prelievo e rimane tutto in memoria. Per ciascun simbolo si fa attenzione alla grammatica del tipo considerato, sia nelle spaziature che nella scrittura numerica (ad esempio *operator>> (double& val)*, con i bit di bad o fail se fallisce prelievo e/o input).

L’operazione svolta è il parsing, che interpreta la sequenza di byte con un algoritmo apposito usato dal parser.

Ad esempio sotto:

```
istream& operator>>(istream& in, Punto& p){
    char cc; in >> cc;
    if(cc=='q') return in;
    if(cc != ' '){ in.clear(ios::failbit); return in; }
    else{
        in >> p.x;
        if(!in.good()) { in.clear(ios::failbit); return in; }
        in >> cc;
        if(cc != ','){ in.clear(ios::failbit); return in; }
        else{
            in >> p.y;
            if(!in.good()) { in.clear(ios::failbit); return in; }
            in >> cc;
            if(cc != ' '){ in.clear(ios::failbit); return in; }
        }
    }
    return in;
}
```

Invoco l’algoritmo e mi affido allo stream ed ai bit per capire se l’input ha avuto successo e la relativa posizione.

Quindi controlla che rispetto a (double, double) gli spazi non siano dopo gli stessi di input e controlla passo per passo se c’è correttezza negli input su ogni carattere; in effetti non è facile scrivere un buon algoritmo di parsing, dato che potrebbe non essere robusto ed adattabile alle singole situazioni, infatti è il caso dell’algoritmo sopra.

A questo punto diciamo che *cin* è parte di *istream*; nell’esempio sotto:

```
#include "Punto.h"
using std::cin; using std::cout;

int main() {
    Punto p;
    cout << "Inserisci un punto nel formato (x,y) ['q' per uscire]\n";
    while(cin.good()) { // while(stato == 0)
        cin >> p;
        if(cin.fail()) {
            cout << "Input non valido, ripetere!\n";
            cin.clear(ios::goodbit);
            char c=0;
            // 10 è il codice ASCII del carattere newline
            while(c!=10) { cin.get(c); } // svuota cin, get() per input binario
            cin.clear(ios::goodbit);
        }
        else cin.clear(ios::eofbit); // stato 1
    }
}
```

Io controllo il bit di end of file, tale che se non è ancora stato settato ci sono degli errori sugli I/O, verificando ciclicamente se la correttezza c’è rispetto ai bit e relativo controllo (parte sopra con gli operatori di tipo) e poi il clear che setta il bit corretto.

Passiamo al tipo *ostream*, che agisce in un modo simile con *operator<<*; di questo ne esistono due varianti, sia relativa ai byte che alle singole stringhe.

```
orario sommaDueOrari() {
    orario o1,o2;
    try { cin >> o1; } // può sollevare eccezioni
    catch (err_sint)
    {cerr << "Errore di sintassi"; return orario();}
    catch (fine_file)
    {cerr << "Errore fine file"; abort();}
    catch (err_ore)
    {cerr << "Errore nelle ore"; return orario();}
    catch (err_minuti)
    {cerr << "Errore nei minuti"; return orario();}
    catch (err_secondi)
    {cerr << "Errore nei secondi"; return orario();}
    try { cin >> o2; } // può sollevare eccezioni
    catch (err_sint)
    {cerr << "Errore di sintassi"; return orario();}
    catch (fine_file)
    {cerr << "Errore fine file"; abort();}
    catch (err_ore)
    {cerr << "Errore nelle ore"; return orario();}
    catch (err_minuti)
    {cerr << "Errore nei minuti"; return orario();}
    catch (err_secondi)
    {cerr << "Errore nei secondi"; return orario();}
    return o1+o2;
}
```

Ad esempio: `ostream& std::operator<<(ostream&, char);`
`ostream& std::operator<<(ostream&, const char* s);`

Per esempio la rappresentazione in formale testuale si passa direttamente alla rappresentazione in memoria (eseguita in formato binario), perché tratto i byte senza nessuna particolare interpretazione. Nel caso dell'input binario, effettuato carattere per carattere, viene normalmente effettuato con metodi di *get*, sempre effettuato con metodi di *istream*, le quali non falliscono finché continuano a prelevare byte, metodo *put*, che agisce sui singoli caratteri.

A noi ciò che interessa sono gli stream singoli, quindi *ifstream*, *ofstream* e *fstream*, per i quali sono intesi costruttori, presenti a basso livello (ad es. `const char*`) e la modalità di apertura (*openmode*), per cui ne esistono 7, ad esempio apertura in lettura o scrittura, spostamento a EOF, erase file, ecc. A questo, per esempio, l'*eof* può determinare dove finisce la testina di I/O, capendo in che posizione abbiamo iniziato a leggere, finito di leggere oppure dove posizionarsi per acquisire input successivi, facendo un prelievo binario dei singoli bit e appendendo (append) i bit mentre li sto leggendo.

Utile anche in ottica progetto, sono gli stream associati a stringhe (ad esempio le coordinate grafiche, facendo il parsing dell'input dell'utente) risiedenti in RAM. Analogamente a prima, le classi da utilizzare sono *istringstream*, *ostringstream* e *stringstream*.

Il file header di riferimento è `<sstream>`; se l'input proviene da GUI, non essendo un input tradizionale, ciascuno di questi deve essere parsato (elaborato) correttamente.

Per la descrizione delle classi vedere libro pg 252.

Gestione delle eccezioni

Le eccezioni vengono gestite parallelamente alla gestione della memoria eseguita dal call stack, tramite le keywords *throw*, che lancia solitamente una eccezione di tipo definito dall'utente, oppure in uno specifico blocco di programma tramite *try* e *catch*, che non fa altro che catturare l'eccezione di un certo tipo ed è in grado di gestire la situazione eccezionale. Altro esempio è il comando *abort*, che genera il segnale di terminazione anomala uscendo quindi dall'esecuzione di un programma.

All'interno di un programma è importante implementare anche il *parsing*, quindi l'elaborazione degli input dei singoli tipi a seconda che vengano letti dei caratteri piuttosto che altri; a tale scopo vengono implementate delle classi di eccezione dell'errore.

Per esempio:

```
istream& operator>>(istream& is,
                   orario& o) {
    char c; string::size_type pos;
    string cifre("0123456789");
    int ore, minuti, secondi;
    if (!is >> c) throw fine_file();
    // prima cifra ore
    pos = cifre.find(c);
    if (pos == string::npos)
        throw err_sint();
    ore = pos;
    if (!is >> c) throw fine_file();
    if (c != ':') {
        // seconda cifra ore
        pos = cifre.find(c);
        if (pos == string::npos)
            throw err_sint();
        ore = ore * 10 + pos;
        if (ore > 23) throw err_ore();
        if (!is >> c)
            throw fine_file();
    }
    // ore lette, c deve essere ':'
    if (c != ':') throw err_sint();
    if (!is >> c) throw fine_file();
    // prima cifra minuti
    pos = cifre.find(c);
    if (pos == string::npos)
        throw err_sint();
    minuti = pos;
    if (!is >> c) throw fine_file();
    if (c != ':') {
        // seconda cifra minuti
        pos = cifre.find(c);
        if (pos == string::npos)
            throw err_sint();
        minuti = minuti * 10 + pos;
        if (minuti > 59) throw err_minuti();
        if (!is >> c) throw fine_file();
    }
    // minuti letti, c deve essere ':'
    if (c != ':') throw err_sint();
    if (!is >> c) throw fine_file();
    // prima cifra secondi
    pos = cifre.find(c);
    if (pos == string::npos)
        throw err_sint();
    secondi = pos;
    if (is && cifre.find(c) != string::npos) {
        // per cin, is=0 con Ctrl-D
        // seconda cifra secondi
        pos = cifre.find(c);
        secondi = secondi * 10 + pos;
        if (secondi > 59) throw err_secondi();
    }
    // ho letto i secondi
    else if (is) // carattere non cifra
        is.putback(c);
    o.sec = ore*3600 + minuti*60 + secondi;
    return is;
}
```

Si esamina quindi il parsing caso per caso; per esempio nel caso dei caratteri dei file, determinando subito la fine del file a seconda che manchino i singoli caratteri oppure gli errori di fine stringa a seconda che manchino i singoli caratteri; il formato di gestione definito è in effetti abbastanza robusto, perché gestisco ogni errore a seconda che appaiano.

Nel caso sopra si ha una situazione non ottimale, con una singola lista di chiamate *catch* all'interno dei *try*; problema grosso la ripetizione di codice realizzato più volte. I *throw* possono lanciare un errore di qualsiasi tipo; attenzione a non mettere i *throw* dentro i *try*.

Se il *throw* è collocata in un blocco *try* nel corpo della funzione, l'esecuzione passa subito al *catch*; se si trova un type match per l'eccezione, essa viene catturata e utilizzata; se non si trova, si passa all'elemento next all'interno del call stack. Si nota quindi come la pratica venga gestita in modalità topdown; se non si trova nulla anche nel main e *abort*, si lancia il segnale di terminate e quindi *abort*.

È possibile eventualmente rilanciare eccezioni dentro altri *catch*, addirittura.

Attenzione all'utilizzo delle risorse, le quali devono essere allocate in una loro area per poi essere rilasciate in caso di eccezione; il pattern è questo, una volta finito l'utilizzo di una certa risorsa, la si rilascia e gestisce eventuali eccezioni di apertura, usando la solita istruzione *delete* liberando quindi la memoria.

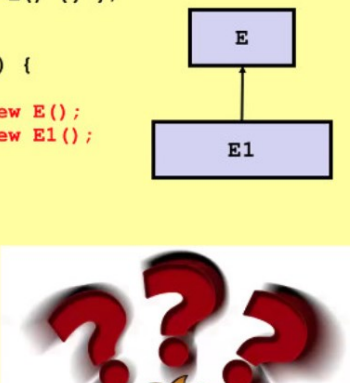
Match del tipo delle eccezioni (intendendo T come tipo ed E come eccezione):

- il tipo T è uguale al tipo E
- il tipo E è un sottotipo di T (gestione delle eccezioni in una gerarchia di tipi polimorf); valida comunque sia nel caso di puntatore che riferimenti tra tipo e sottotipo nel modo descritto
- T di tipo `void*` ed E un tipo puntatore generico
- Non vengono applicate conversioni implicite

```
class E { public: virtual ~E() {} };
class E1: public E {};

void modify(vector<int>& v) {
    ...
    if(v.size()==0) throw new E();
    if(v.size()==1) throw new E1();
    ...
}

void G(vector<int>& v) {
    try{
        modify(v);
    }
    catch(E* p) {...}
    catch(E1* q) {...}
}
```



Attenzione che nel caso a fianco si ha un errore logico, quindi il compilatore non si chiama.

Comportamenti da seguire:

Comportamenti tipici di una clausola catch sono i seguenti:

- rilanciare un'eccezione
- convertire un tipo di eccezione in un altro, rimediando parzialmente e lanciando un'eccezione diversa
- cercare di ripristinare il funzionamento, in modo che il programma possa continuare dall'istruzione che segue l'ultima catch
- analizzare la situazione che ha causato l'errore, eliminarne eventualmente la causa e riprovare a chiamare la funzione che ha causato originariamente l'eccezione
- esaminare l'errore ed invocare `std::terminate()`

La specifica delle eccezioni esplicita è stata poi tolta da C++ 11; è stata poi tolta per vari motivi, tra cui il fatto che non gestiscono quelle a compile-time e si ha un overhead in tempo reale; anche nei template non viene utilizzata.

Parlando di Qt per esempio sappiamo che le eccezioni all'inizio non c'erano; tuttora nel modulo GUI di Qt non è disponibile, ma ovviamente in C++. Per la correttezza dei programmi non è il massimo usare eccezioni.

CAPITOLO 9 – “Libreria Qt”

Qt è una libreria per lo sviluppo di interfacce utente grafiche. È usata per lo sviluppo di software ad ampissima diffusione quali Photoshop, Skype, VirtualBox e da multinazionali quali Google, HP, Samsung etc. Una componente utile è QT Designer, che genera molto codice automatico e per questo non è particolarmente consigliabile. Qt è formato completamente ad oggetti e la gestione dei singoli oggetti è dato da MOC (Meta Object Compiler), il quale gestisce tutti i segnali (parte fondamentale) agli slot rilevati da *Qmake*.

Esempio standard di applicazione grafica è la calcolatrice, GUI minimale con pulsanti ed etichette intuitivi tali da offrire una minima funzionalità all'utente. Vari eventi sono il click, il muovere il puntatore del mouse, ecc. Tutto ciò deve essere gestito nel modello **Segnali/Slot** di Qt (intendendo per slot la funzione che viene attivata in una determinata circostanza).

9.1 Introduzione Qt

Qt include un insieme di classi che offrono dei tipi di dato alternativi a quelli del C++ standard, tra i quali: *QChar*, *QString*, *Qvector<T>*, *QVectorIterator*, *QList<T>*, *QLinkedList<T>*, *QSet<T>*, *QMap<Key,T>*, *QDir*, *QFile*, *QTextStream*, *QDate*, *QTime*, *QPoint* etc

QObject è la classe alla base del modello ad oggetti di Qt, in cui esistono relazioni di parentela/fratellanza (quindi tipi e sottotipi).

Il meccanismo **Signal/Slot** utilizzato per la comunicazione tra gli oggetti delle classi Qt è la caratteristica che differenzia Qt da altri framework per

lo sviluppo di GUI. Quando cambia lo stato di qualche componente di una GUI, detto **widget**, spesso si vuole che ciò provochi la notifica di tale cambiamento ad un altro widget. In generale si vuole che oggetti di qualsiasi tipo siano in grado di comunicare tra loro.

Un **Widget** emette un particolare segnale quando si verifica un corrispondente evento particolare nel widget stesso. Questi hanno molti segnali predefiniti ed è inoltre possibile definire widget sottotipo per aggiungere ulteriori segnali.

Tutte le classi che ereditano da QObject o dalle sue sottoclassi possono contenere segnali e slot.

Uno **Slot** è una funzione che viene chiamata in risposta ad un particolare segnale emesso. I widget molto spesso hanno molti slot predefiniti ed è pratica comune definire widget sottotipo per aggiungere altri slot. Gli Slot possono essere utilizzati per ricevere segnali, ma sono anche dei metodi ordinari di una classe. Uno Slot non sa se ha dei segnali ad esso collegati.

I **Signal** vengono emessi dagli oggetti quando essi cambiano stato in un modo che può essere di interesse per altri oggetti che intendono reagire a tale cambiamento di stato. L'oggetto non fa altro che comunicare il suo cambiamento di stato, anche se non sa se qualche altro oggetto sta ricevendo i suoi segnali. Ciò permette quindi di avere un netto ed efficace incapsulamento dell'informazione.

Segnali e Slot sono debolmente accoppiati: un oggetto che emette un segnale non conosce né si preoccupa degli slot che eventualmente riceveranno tale segnale. Il meccanismo di connessione tramite connect tra signal e slot assicura che se si collega un segnale ad uno slot, lo slot sarà chiamato con i parametri che caratterizzano il segnale al momento giusto.

Ciò assicura che le componenti Qt siano realmente indipendenti tra loro.

NB: E' possibile collegare un qualsiasi numero di segnali per un singolo slot, ed un segnale può essere collegato ad un qualsiasi numero di slot.

9.2 Hello World

```
#include <QApplication>
#include <QLabel>
#include <QPushButton>

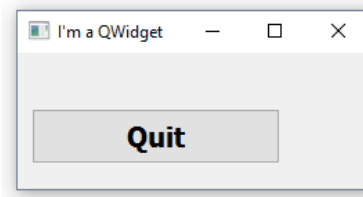
int main(int argc, char* argv[]){
    QApplication app(argc, argv);
    QPushButton hello("Hello world!");
    hello.show();
    return QApplication::exec();
}
```

- Il file header `<QApplication>` va sempre incluso, ogni programma Qt deve costruire un oggetto `QApplication`, al cui costruttore sono passati gli argomenti `argc` e `argv` del metodo `main` del programma.
- Il file header `<QLabel>` corrisponde al widget Qt usato dal programma e che permette di usare testo o un'immagine.
"hello" è un widget di tipo `QLabel` creato tramite una chiamata al costruttore `QLabel(const QString t, QWidget* parent = 0, Qt::WindowFlags f = 0)`. La stringa "Hello World" è convertibile al tipo `QString`. Il widget "hello" non ha parent widget, lui stesso è una window con frame e title bar.
- L'invocazione `hello.show()` rende visibile un widget che altrimenti non sarebbe visibile di default.
- L'invocazione del metodo statico `QApplication::exec()` cede il controllo dal main a Qt.

9.3 Segnali e Slot

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

int main(int argc, char* argv[]){
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("I'm a QWidget");
    window.resize(200,120);
    QPushButton quit("Quit", &window);
    quit.setFont(QFont("Times", 15, QFont::Bold));
    quit.setGeometry(10, 40, 180, 40);
    QObject::connect(&quit, SIGNAL(clicked()),&app, SLOT(quit()));
    window.show();
    return app.exec();
}
```



- **<QWidget>** è la classe base di tutti i widget. Una QWidget è un atomo di una GUI, esso riceve eventi dal sistema con cui interagisce (mouse, keyboard, touch screen etc) e rappresenta se stesso sullo schermo (la posizione iniziale è controllata dal sistema).
Una QWidget è detenuta dal suo cosiddetto parent. (una QWidget senza parent viene detta independent window)
- L'invocazione di `setWindowTitle()` che è un public slot di QWidget permette di definire il titolo di "window" mentre `resize()` ridimensiona window.
- "quit" è un **<QPushButton>**, un widget con funzionalità di pulsante. Inoltre quit ha window come parent ⇒ quit è figlio di window.
Un figlio è sempre mostrato nell'area del suo parent (per default alla posizione (0,0) top-left).
- `quit.setFont()` definisce le proprietà delle font del pulsante quit;
- `quit.setGeometry(x,y,w,h)` definisce (x,y) come coordinate del top-left corner di quit e (w,h) come dimensione base ed altezza di quit.
- L'invocazione `connect()` è un metodo statico di **QObject** (classe base di ogni widget) che stabilisce una connessione tra due QObject.
Ogni QObject e quindi ogni QWidget può avere dei Signal per spedire messaggi e degli Slot per riceverli.
⇒ `SIGNAL(clicked())` emesso da quit viene quindi connesso allo `SLOT(quit())` di app.
In questo caso, grazie alla `connect()` otteniamo l'effetto che quando si clicca il pulsante "quit" ⇒ app termina.
- `window.show()` invoca il metodo `show()` anche su tutti i figli di window.

La precedente QWidget può anche essere progettata come una classe definita dal programmatore che eredita QWidget nel seguente modo:

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

class MyWidget : public QWidget{
public:
    MyWidget(QWidget *parent = 0);
};

#endif
```



```
#include "widget.h"
#include <QApplication>
#include <QFont>
#include <QPushButton>

MyWidget::MyWidget(QWidget *parent) : QWidget(parent){
    setWindowTitle("I'm a Widget");
    resize(200,120);
    QPushButton* quit = new QPushButton(tr("Quit"),this);
    quit->setFont(QFont("Times",15,QFont::Bold));
    quit->setGeometry(10,40,180,40);
    connect(quit, SIGNAL(clicked()),qApp, SLOT(quit()));
}
```

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]){

    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();

    return app.exec();
}
```

- Qui definiamo un costruttore con argomento QWidget parent il cui valore di default 0 significa che è a top-level (senza parent).

9.4 Layout Manager

```
#ifndef WIDGET2_H           widget2.h
#define WIDGET2_H

#include <QWidget>

class MyWidget2 : public QWidget{
public:
    MyWidget2(QWidget *parent = 0);
};

#endif
```



```
#include "widget2.h"                widget2.cpp
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QLCDNumber>
#include <QSlider>
#include <QVBoxLayout>

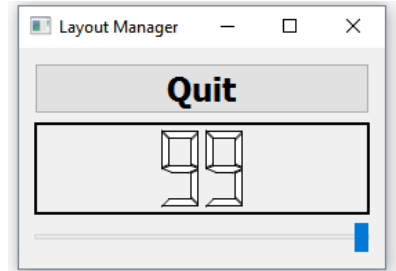
MyWidget2::MyWidget2(QWidget *parent) : QWidget(parent){
    setWindowTitle("I'm a Widget");
    QPushButton* quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));
    QLCDNumber* lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);
    lcd->setPalette(Qt::black);
    QSlider* slider = new QSlider(Qt::Horizontal);
    slider->setRange(0,99);
    slider->setValue(0);
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
    QVBoxLayout* layout = new QVBoxLayout;
    layout->addWidget(quit);
    layout->addWidget(lcd);
    layout->addWidget(slider);
    setLayout(layout);
}
```

```
#include "widget2.h"
#include <QApplication>

int main(int argc, char *argv[]){

    QApplication app(argc, argv);
    MyWidget2 widget;
    widget.show();

    return app.exec();
}
```



- **<QLCDNumber>** è un widget che mostra numeri, in questo caso 2 con un look-and-feel LCD;
- **<QSlider>** è un widget che permette di selezionare un valore intero tra un range di valori.
- Con la seconda connect il segnale `valueChanged(int)` del widget slider viene connesso allo slot `display(int)` del widget lcd in modo che quando slider emette un segnale `valueChanged()` quando cambia il suo valore allora lo slot `display` viene invocato.
- **<QVBoxLayout>** è un layout manager(LM) il quale gestisce la geometria dei figli di un widget anche in caso di ridimensionamento.
Un `QLayout`(supertipo di `QVBoxLayout`) non è sottotipo di `QWidget` e quindi un LM non ha mai un parent.
Con il metodo `addWidget()` vengono aggiunti dei widget al controllo di un LM;
Con il metodo `setLayout(layout)` viene installato layout come LM di this e rende layout figlio di this.
Tutti i widget sotto controllo di layout diventano figli di this.

9.5 Connessioni tra Segnali

```
#include "lcdrange.h"
#include <QLCDNumber>
#include <QSlider>
#include <QVBoxLayout>

LCDRange::LCDRange(QWidget* parent) : QWidget(parent){
    QLCDNumber* lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);
    slider = new QSlider(Qt::Horizontal);
    slider->setRange(0,99);
    slider->setValue(0);
    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));
    connect(slider, SIGNAL(valueChanged(int)),
            this, SIGNAL(valueChanged(int)));
    QVBoxLayout* layout = new QVBoxLayout;
    layout->addWidget(lcd);
    layout->addWidget(slider);
    setLayout(layout);
}

int LCDRange::value() const{
    return slider->value();
}

void LCDRange::setValue(int value){
    slider->setValue(value);
}
```

```
#ifndef LCDRANGE_H
#define LCDRANGE_H

#include <QWidget>

class QSlider;

class LCDRange : public QWidget{
    Q_OBJECT
public:
    LCDRange(QWidget* parent = 0);
    int value() const;
public slots:
    void setValue(int value);
signals: //implicitamente protected
    void valueChanged(int newValue);
private:
    QSlider* slider;
};

#endif // LCDRANGE_H
```

```
#include "mywidget.h"
#include <QApplication>

int main(int argc, char *argv[]){

    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();

    return app.exec();
}
```

```
#include "mywidget.h"
#include "lcdrange.h"

#include <QApplication>
#include <QFont>
#include <QGridLayout>
#include <QPushButton>
#include <QWidget>

MyWidget::MyWidget(QWidget* parent) : QWidget(parent){
    QPushButton* quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    QGridLayout* grid = new QGridLayout;
    LCDRange* previousRange = 0;
    for(int row = 0; row < 3; ++row){
        for(int column = 0; column < 3; ++column){
            LCDRange* lcdRange = new LCDRange;
            grid->addWidget(lcdRange, row, column);
            lcdRange->setPalette(Qt::black);
            if(previousRange)
                connect(lcdRange, SIGNAL(valueChanged(int)),
                        previousRange, SLOT(setValue(int)));
            previousRange = lcdRange;
        }
    }
    QVBoxLayout* vlayout = new QVBoxLayout;
    vlayout->addWidget(quit);
    vlayout->addLayout(grid);
    setLayout(vlayout);
}
```

```
#ifndef WIDGET_H
#define WIDGET_H

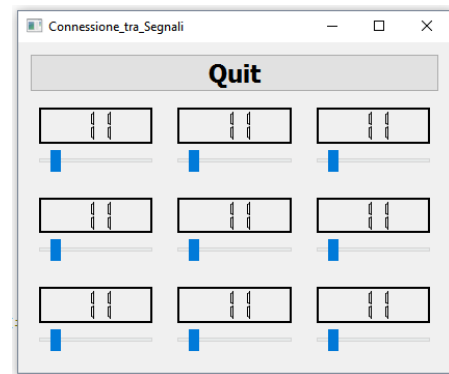
#include <QWidget>

class MyWidget : public QWidget{
public:
    MyWidget(QWidget *parent = 0);
};

#endif // WIDGET_H
```

- La macro Q_OBJECT deve essere inclusa in tutte le classi che definiscono dei segnali o delle slot;
- La classe LCDRange:
 - include come public slot il metodo void setValue(int) che definisce il valore intero del QSlider slider;
 - dichiara come signal(sempre implicitamente protected);
- Nel costruttore di LCDRange è inclusa la connect che permette di fare una connessione tra signal in modo che quando uno slider predefinito void valueChanged(int) viene anche emesso il signal valueChanged(int) dall'oggetto this di tipo LCDRange;

■ Nel costruttore di MyWidget all'interno del for è inclusa una connect che permette di connettere il signal void valueChanged(int) di ogni LCDRange costruito con il public slot void setValue(int) del LCDRange costruito all'iterazione precedente. In questo modo siamo riusciti a creare una catena di segnali e slot tra i 9 LCDRange della GUI in modo tale che una variazione del valore di qualche LCDRange della griglia cambia il valore di tutti i LCDRange che lo precedono in griglia.



C++ 11 e altre considerazioni

In C++ 11, le eccezioni avvengono a runtime e non a compile time e non erano compatibili con i template.

Si ha l'inferenza di tipo, tramite la keyword *auto*, piuttosto che ricordarsi dei tipi molto lunghi da scrivere.

In questo modo si recupera dinamicamente il tipo dal ritorno dell'espressione come r-valore

(ad esempio: `auto ci = vi.begin()` su un vettore costante di tipo `<int>` deduce che si tratta di un iteratore e nel qual caso ad un intero costante.

Altra keyword utile è *decltype*, che determina staticamente il tipo di una espressione.

Esempio d'uso:

```
int x=3; decltype(x) y=4;
```

Per poter inizializzare contenitori di dimensione generale si possono utilizzare le parentesi graffe {}

Esempio d'uso l'array dinamico:

```
int *a = new int[3] {1,2,0};
```

Si tratta quindi niente altro che di una costruzione di copia di un array.

Si può fare anche nei contenitori STL nel seguente modo:

```
std::vector<string> vs = {"first", "second", "third"};
```

```
std::map<string, string> singers = {"federer", "3470123456"}, {"Roger", "3489875245"};
```

Quali sono le operazioni con comportamento di default? 4, costruttore di default e di copia, distruttore ed assegnazione.

Con esse può esservi utile la keyword *default*, ad esempio:

```
class A {
public:
    A(int) {} // costruttore ad 1 argomento
    A() = default; // costruttore altrimenti non disponibile
    virtual ~A() = default; // distruttore virtuale standard
};
```

Sto solo dicendo che uso una certa struttura nella sua versione di default senza altre informazioni.

Analogamente, ho anche la stessa cosa con un'altra keyword, chiamata *delete*. Essa serve a mantenere la funzionalità di alcuni oggetti, ma disabilitandoli all'esterno. Saranno ugualmente invocabili, ma li cancella a tutti gli effetti.

```
class NoCopy {
public:
    NoCopy& operator=(const NoCopy& ) = delete;
    NoCopy (const NoCopy&) = delete;
};

int main() {
    NoCopy a,b;
    NoCopy b(a); // errore in compilazione
    b=a; // errore in compilazione
}
```

Una classe importante che non rende disponibili distruzione e costruzione di copia è *typeinfo*. Utilizziamo la stessa keyword in cui non permetto conversioni di alcun tipo (esempio utile di questo sono le eccezioni, dove ammetto solo un tipo esatto e non converto nulla).

```
static void fun(double) {}
template <class T> static void fun(T) = delete;
// NESSUNA CONVERSIONE A DOUBLE PERMESSA
```

Con l'uso della keyword detta, non permettiamo la compilazione regolarmente e risulta un errore come qui sotto:

```
int main() {
    int a=5; float f=3.1;
    OnlyDouble::fun(a); // ILLEGALE: use of deleted function with T=int
    OnlyDouble::fun(f); // ILLEGALE: use of deleted function with T=float
}
```

Parliamo della keyword *override*, che permette l'overloading non solo per funzioni già effettivamente definite ma anche per le funzione astratte (o virtuali pure). Essa si accompagna con la keyword *final*, presa da Java, dove ogni metodo è virtuale (con late binding, che risulta costoso). Java permette di dire che, ad un certo punto, non vi sono altri overriding di una particolare funzione, facendo quindi una sorta di binding statico dopo X binding dinamici.

Altro esempio utile sono i puntatori nulli (`nullptr` o `0`, letterale di tipo `int` e invoca `f(int)`, come sotto:

```
void f(int);
void f(char*);

int main() {
    f(0); // quale f invoca? invoca f(int)
}
```

Il tipo `nullptr` è definito dallo standard (`std::nullptr_t`), implicitamente convertibile a qualsiasi puntatore e a dei `bool`.

```
void f(int);
void f(char*);

int main() {
    f(nullptr); // quale f invoca? invoca f(char*)
}
```

```
const char* pc = str.c_str();
if (pc != nullptr) std::cout << pc << endl;
```

Altro esempio di uso sopra, dove stavolta viene chiamato un *char*.

Un costruttore nella sua lista di inizializzazione può invocare un altro costruttore della stessa classe, meccanismo noto come *delegation* e disponibile in alcuni linguaggi come Java. Questa cosa è definita come *delegation*, dove vengono definiti dei valori di default per la lista di inizializzazione della serie di valori che si hanno in quel momento. Esempio pratico: C(): C(0,0)

Interessanti anche degli oggetti a cui si fa compiere particolari azioni da oggetti che rappresentano delle funzioni, quindi che devono essere invocati con dei parametri e ritornare un tipo. Questo fa riferimento al paradigma di programmazione funzionale.

Ad esempio:

```
FunctorClass fun;  
fun(1,4,5);
```

È possibile fare questo con l'overloading di `operator()`, operatore chiamata di funzione, invocando un numero qualsiasi di parametri di qualsiasi tipo e ritornare un tipo qualunque. Al di là dell'esempio sotto, con questo overloading, si scrivono *funzioni di alto livello*.

Altro esempio:

```
class FunctorClass{  
private:  
int x;  
public:  
FunctorClass(int n): x(n) {}  
int operator() (int y) const {return x+y;}  
};  
  
int main(){  
FunctorClass sommaCinque(5);  
cout << sommaCinque(6);  
}
```

L'operazione di moltiplicazione per 2 viene quindi normalmente riutilizzato, come nel caso sotto. Il funtore si applica in tutti i tipi di contenitori. Ad ogni elemento applica `op` e memorizza il valore ritornato da ogni applicazione di `op` di contenitore che inizia da `result`. Ciò è equivalente all'utilizzo di un template e di relativi overload.

```
class MoltiplicaPer{  
private:  
int factor;  
public:  
MoltiplicaPer(int x): factor(x) {}  
int operator() (int y) const {return factor*y;}  
};  
  
int main(){  
vector<int> v;  
v.push_back(1); v.push_back(2); v.push_back(3);  
cout << v(0) << " " << v(1) << " " << v(2) << endl;  
std::transform(v.begin(), v.end(), v.begin(), MoltiplicaPer(2));  
cout << v(0) << " " << v(1) << " " << v(2) << endl;  
}
```

```
class UguaileA{  
private:  
int number;  
public:  
UguaileA(int n): number(n) {}  
bool operator() (int y) const {return x==number;}  
};  
  
template <class Functor>  
vector<int> find_matching(const vector<int>& v, Functor pred){  
    vector<int> ret;  
    for(vector<int>::const_iterator it = v.begin(); it<v.end; ++it)  
        if(pred(*it)) ret.push_back(*it);  
    return ret;  
}
```

Citavamo prima la funzione *transform*, che usa un contenitore in forma R/W, ma anche il template `for_each`, definito sempre da `std::for_each`, che utilizza iteratori al primo/ultimo elemento e funzione unaria (InputIterator first, InputIterator last, UnaryFunction f). Esse fanno parte di `<algorithm>`

Una lambda espressione è un funtore che prende un letterale, funtore definito *al volo*. Esso non è un tipo funtore.

Per esempio:

```
std::for_each(v.begin(), v.end(), [](int x) {cout << fattore*x << " "});
```

Il funtore richiede un intero e non ritorna nulla, quindi tipo void di ritorno. In poche parole risparmia a scrittura di un funtore. Esse sono definite come chisure (closures), che definisce le variabili da usare in una certa lambda-espressione.

Risparmia ad esempio di scrivere, nel caso sopra:

```
Functor f(2);  
std::for_each(v.begin(), v.end(), f);
```

Lambda deriva dal concetto di programmazione funzionale, ad esempio (lambda)xf(x), che astrae il concetto di parametro e funzione.

Altro esempio sono le funzioni anonime locali, ad esempio

```
[]->int {return 3*3;} (freccetta perché ci sta lista vuota di parametri)  
{}(int x, int y) {return x+y;}
```

Esempio d'uso: [x, &y]: variabili per scrittura/lettura (x) e altri in sola lettura (y)

Un piccolo appunto è *this*, che passo in sola lettura ed è solo per valore, perché parte del contesto di invocazione.

Shared pointers

Gestione automatica della creazione/distruzione di oggetti con reference counting (smart pointers).

Per rendere la memoria condivisa e usare il costruttore shared_ptr, si usa il metodo `std::make_shared` (è un equivalente al costruttore in effetti).

Come per i puntatori ordinari, è previsto l'overloading dell'operatore "`->`" e si usa il metodo `use_count()` per tener conto degli oggetti creati.

Sono previsti gli overloading degli operatori di assegnazione (sia nel caso di nullptr e nella conversione a shared_ptr; quando si crea garbage, con questi viene automaticamente deallocato).

Un esempio pratico:

shared_ptr in bolletta

```
class bolletta {  
public:  
    // parte pubblica non cambia  
private:  
    class nodo { // definizione di nodo  
    public:  
        nodo();  
        nodo(const telefonata&, const std::shared_ptr<nodo>&);  
        telefonata info;  
        std::shared_ptr<nodo> next; // smart ptr next  
    }; // fine classe nodo  
  
    std::shared_ptr<nodo> first; // smart ptr first  
};
```

Gestione quindi di una lista concatenata con la classe bolletta, uno dei primi esempi. Si usano modifiche dei costruttori precedenti, costruendo nuovi nodi in maniera condivisa, come:



```
// Metodi di bolletta  
// Le definizioni sono semplici  
  
void bolletta::Aggiungi_Telefonata(const telefonata& t) {  
    first = make_shared<nodo>(t, first);  
}  
  
bool bolletta::Vuota() const {  
    return first == nullptr;  
}
```