INGEGNERIA DEL SOFTWARE

LINGUAGGI OO e C++

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati da me con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it
web: http://www.morpheusweb.it

LINGUAGGI OBJECT ORIENTED

Un linguaggio ad oggetti deve supportare:

- definizione di tipi di dato astratto
- ereditarietà
- polimorfismo
- binding dinamico per l'invocazione dei metodi nei corpi (dynamic dispatching)

EREDITARIETA'

I tipi sono organizzati in una gerarchia

Supponiamo di avere uno stack e di voler contarne gli elementi. Potremmo riscrivere la classe oppure modificare quella esistente o risolver il problema lato client ma tutte le soluzioni sono poco pulite. La soluzione è usare l'ereditarietà.

Counting_Stack è uno Stack

Baso il nuovo tipo che mi serve sull'implementazione originale. I sottotipi possono avere delle funzioni in più e ridefinire le funzioni virtuali

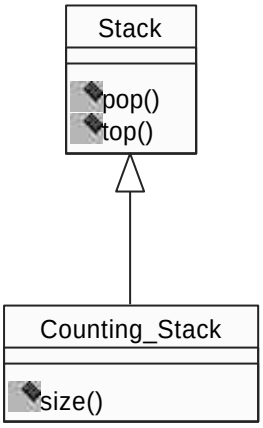
```
class Stack {
public:
     Stack( ); {top = 0;};            //costruttore
     void push(int) {s[top++] = i;};
     int pop( ) {reutrn s[--top];};
protected:
     int top;
private:
     int s[100];
};

class Counting_Stack : public stack {
public:
     int size( ) {return top;}        //eredita da stack
};
```

Attributi **protected**: sono visibili alle sottoclassi ma non all'esterno, attributi **private**: visibili alla sola classe.

In counting_stack ho gli stessi attributi e metodi di stack ed in più ho il metodo size() che ritorna il numero di elementi.

In UML:



L'ereditarietà definisce una relazione "**IS A**". Ogni counting_stack è uno stack. E' un sottoinsieme perché vi posso fare ulteriori operazioni

POLIMORFISMO

E' legato all'ereditarietà. Si ha quando c'è tipizzazione dinamica.

T classe
T' sottoclasse
T' eredita da T

Se a può riferire ad un oggetto di T → può farlo anche a T' (no viceversa)

COERCION (Se cerco di fare il contrario)

Esempio:
Un riferimento a stack può riferire anche a counting_stack ma non viceversa.

```
Stack *sp = new Stack;                    //puntatore a stack
Ccounting_Stack *sp = new Counting_Stack;          //puntatore a counting_stack

sp = csp;  //lecito

csp = sp;  //ERRORE
```

Non lo consetiamo perché non si può fare type checking statico.

`sp = csp;` //csp è contenuto in sp, csp IS A sp

`csp = sp;` //non è detto che lo sia, potrei assegnare a csp un elemento che non necessariamente è un csp, ed avrei potenzialmente un errore di tipo a runtime.

Funziona così solo per oggetti creati dinamicamente.

> **NOTA:**
>
> TIPO STATICO di un riferimento → tipo con cui è stato dichiarato
>
> TIPO DINAMICO di un riferimento → tipo che la variabile o l'oggetto assume in un certo punto a run-time.
>
> Nei linguaggi statici coincidono, in quelli che supportano il polimorfismo possono essere diversi.

Se l'oggetto è stato creato in modo statico

```
Stack S = Stack();
```

Senza la new, l'oggetto non viene allocato nello heap, ma nello stack e non valgono le regole del polimorfismo bensì quelle di conversione di tipo.

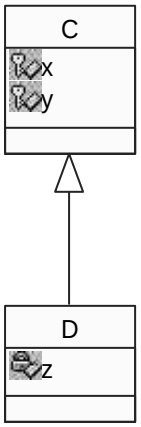
```
Counting_Stack cs;
```

Se faccio

```
s = cs;
```

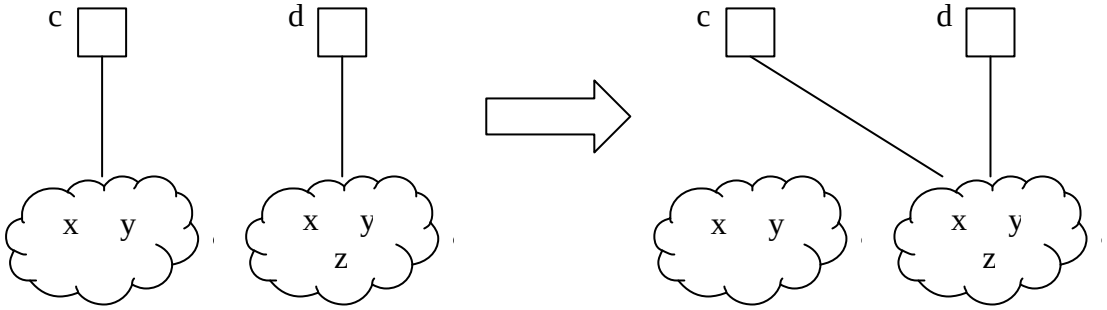
cs viene convertito in stack e si perdono gli elementi in più.

Esempio:




```
C c = new C();
D d = new D();
```

posso fare c = d;

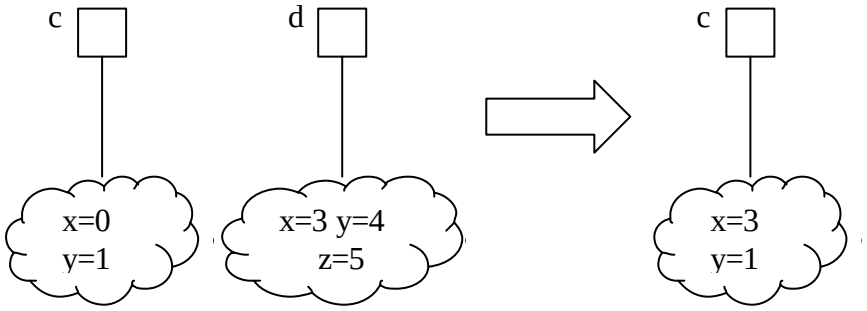


c vede anche z.

se invece faccio:

```
C c = C();
D d = D();
```

con c = d;

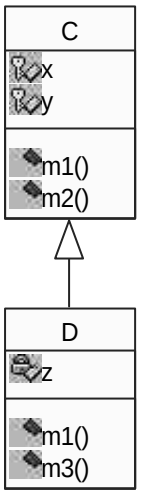


perdo il valore di z.

In C++ il polimorfismo vale **solo** se ho allocato dinamicamente.

BINDING DINAMICO e OVERRIDING

L'ereditarietà fa ereditare anche le operazioni e mi permette anche di ridefinirle.



Eredito m2(), ridefinisco m1() ed ho un nuovo metodo m3().
Ridefinisco un metodo in modo da usare eventuali campi aggiunti in D che in C non avevamo.
Porta ad un'ambiguità:

Il metodo chiamato su un oggetto dipende dal tipo dinamico

Esempio:

Dichiaro a di tipo T, a run-time potrei avere a di tipo T'
T ? tipo statico
T' ? tipo dinamico

```
Stack * sp = new Stack;
Counting_Stack * csp = new Counting_Stack;
...
sp -> push(...);        //stack::push
csp -> push(...);       // counting_stack::push
```

Supponiamo che Counting_Stack ridefinisca il metodo push( ) ➔ **OVERRIDING**

se facciamo:

```
sp = csp;

sp -> push(...);        //QUALE PUSH CHIAMO???
```

Il metodo effettivamente chiamato è quello sul tipo dinamico.

Se invece ho **`sp -> size();`**

Prima di sp = csp;

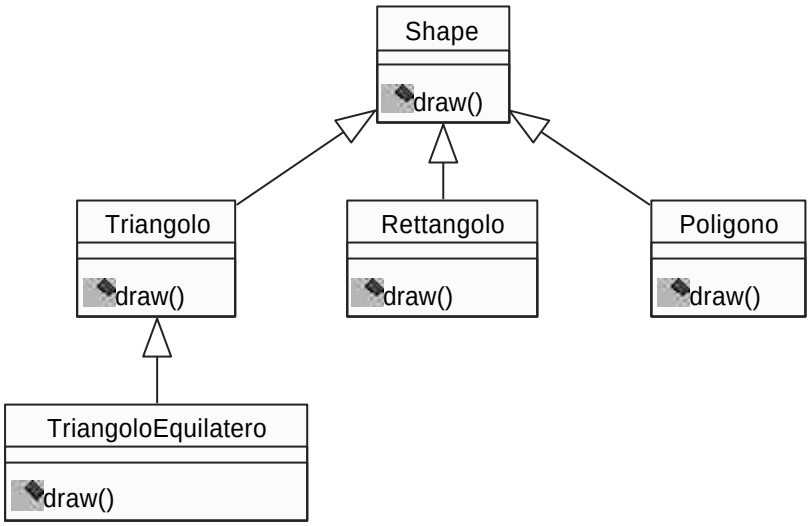
Non la fa poiché non è definita in Stack

Dopo sp = csp;

Se lo permetto non avrei type checking statico, nei linguaggi a tipizzazione forte non lo permetto. Ho dispatching dinamico solo per i metodi già definiti nel tipo statico di riferimento.

FUNZIONI VIRTUALI E BINDING DINAMICO

Esempio:



Ridefinisco draw() nelle varie figure geometriche (o anche altre funzioni di calcolo). In Shape dico solo che c'è il metodo draw(), nelle sottoclassi definisco il codice.

Il default è che il binding delle funzioni è statico. Per farlo dinamico occorre definire le funzioni virtuali. (in JAVA è sempre dinamico)

Esempio:

```
class student {
     public:
          ...
          virtual void print( ) {...};
};

class college_student : public student {
     void print( ) {
     ...                    //funzione specifica per college_student
     }
};

student * s;
college_student * cs;
...
s -> print( );              //chiama student :: print( )

s = cs;                     //va bene

s -> print( );              // chiama college_student :: print( )
```

Definire un metodo VIRTUAL vuol dire che può essere ridefinito nelle sottoclassi.

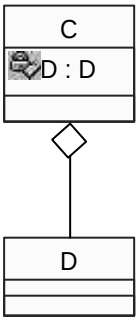
RIASSUNTO

- I tipi sono organizzati in una gerarchia (grafo per l'ereditarietà multipla)
- C'è una relazione di tipo ISA, e l'ereditarietà va usata SOLO quando c'è questa relazione, altrimenti si usano altre cose tipo la composizione

  Esempio:
  Se C non è un D, non possiamo usare

  C
  |
  D

  Ma la composizione




```
class C {

      D d;
      ...

      d.m();
}
```

  Ad esempio se voglio usare ORA nella classe OROLOGIO, non posso usare l'ereditarietà ma devo usare la composizione incapsulando l'ora nella classe orologio

- I sottotipi possono aggiungere delle features e ridefinire i metodi

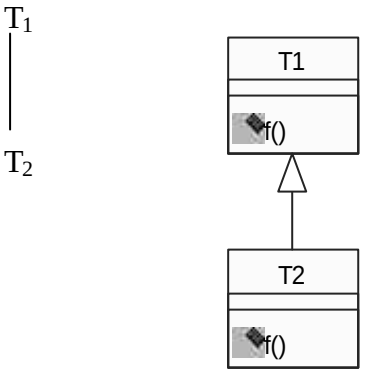
EREDITARIETA' E TYPE SYSTEM

Vogliamo tipizzazione statica, per cui dobbiamo vincolare il modo in cui fare gli assegnamenti.

Quando si ha un etodo m.op(...) che vincoli devo mettere nei parametri sulla relazione di sottotipo per avere tipizzazione forte?

Con i metodi nuovi (definiti nel sottotipo) non ho problemi, poiché non posso chiamarli nel tipo padre.

Il problema nasce con le ridefinizioni



Se un programma definisce T_1 x, il tipo dinamico di x può essere T_2

Ridefinisco un metodo f in T_2 (f metodo di T_1)

Mantenendo la stessa signature non ho problemi (ho l'assegnamente di un sottotipo ad un supertipo)

```
T₁ t₁;
T₂ t₂;

t₁ = t₂;
```

Il tipo dinamico di t1 è quello della sottoclasse.

`t₂.f()` è la funzione della sottoclasse.

Può dare problemi perché potrei ridefinire anche la signature di f() e cambiare il tipo dei parametri che accetto per f().

f : Q1 -> S1

f : Q2 -> S2

In che relazione devono stare Q1 e Q2?

Prima dell'assegnamento:

`t1.f(q1)` q1 di tipo Q1 corretto (prendo il tipo statico)

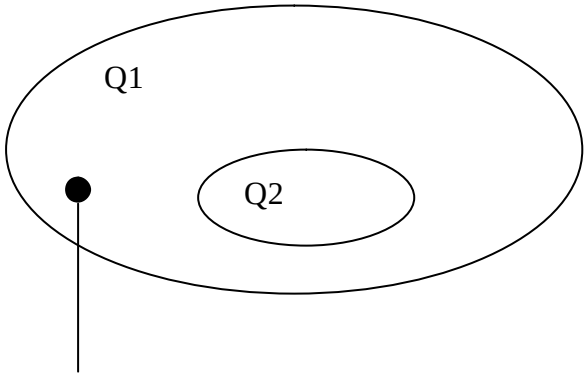
`t1 = t2;`

`t1.f(q1);`

staticamente non sono in grado di dire se è corretto perché dipende dal tipo dinamico di q1 che non conosco.

Se q1 è un sottotipo compatibile con q2, sono a posto, altrimenti no.

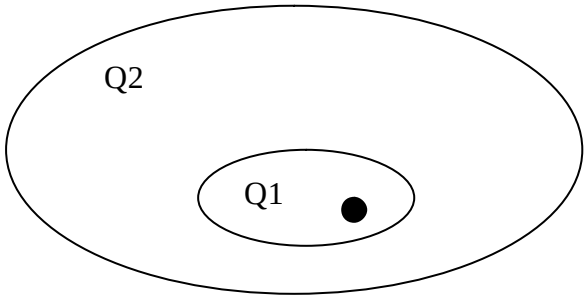
La f definita in T2 si aspetta un parametro di tipo Q2



Non va bene un elemento qui.

La regola è ce il tipo del parametro può essere al massimo uno della superclasse.

Devo avere CONTROVARIANZA nei parametri d'ingresso.



Così sono sicuro che il tipo statico di q1 è sempre contenuto in quello di q2.

E per i **parametri di uscita**?

$S_1 = t_1.f(q_1)$

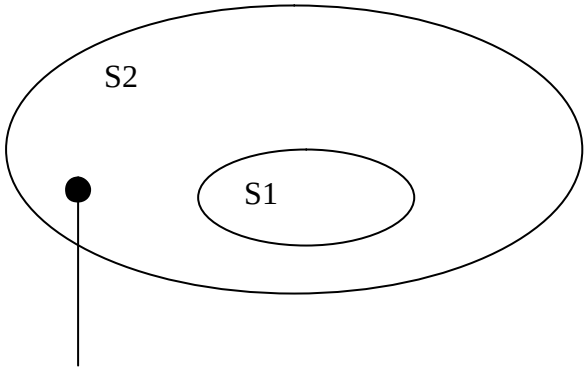
$t_1 = t_2$

$S_1 = t_1.f(q_1)$

Se assumo che

S_2
|
S_1

significa che f può darmi un oggetto che appartiene ad S_2, e devo metterlo nel riferimento di tipo S_1



Potrei avere un oggetto che non è di S1.

Per i parametri di ingresso va applicata la COVARIANZA.

S_1
|
S_2

S2 sta sempre in S1, al massimo può darmi un tipo più piccolo del tipo statico.

RIASSUMENDO: REGOLE DI RIDEFINIZIONE

f : Q1 -> S1
| |
f : Q2 -> S2

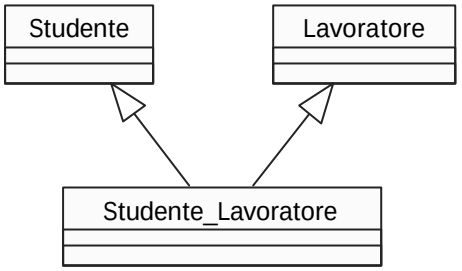
COVARIANZA: per i parametri di uscita
Il tipo dinamico del risultato deve essere definto da una sottoclasse di S1

CONTROVARIANZA: per i parametri d'ingresso
Il tipo dinamico dei q1 deve essere definto da una superclasse di Q1

In genere nei linguaggi si adotta la regola dell'INVARIANZA (è così in C++ e Java)

EREDITARIETA' MULTIPLA

Vuol dire poter ereditare da più classi.

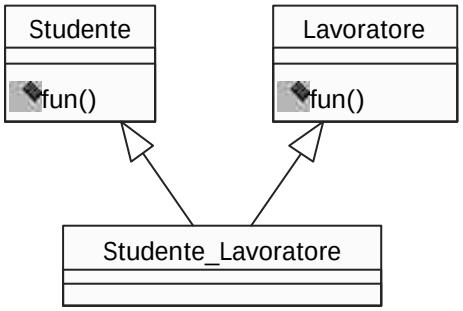


In C++ viene definita nel seguente modo:

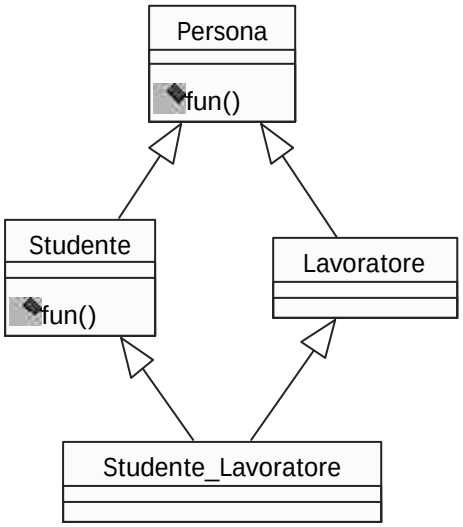
```
class Studente_Lavoratore: public Studente, Lavoratore {
```

Da complessità al run-time support e non tutti i linguaggi la supportano (ad esempio Java).

Un problema è dato dal fatto che due classi potrebbero definire la stessa funzione fun()



Con l'ereditarietà singola, se in Studente_Lavoratore invoco fun() non definita localmente, viene invocato il metodo definito nella superclasse.
Nell'esempio in alto non si capisce qual è la funzione da invocare.
Il discorso è ancora più complesso per strutture a diamante.



In Studente_Lavoratore ho due possibilità: fun() di Persona e fun() di Studente.

In C++ non è possibile invocare il metodo senza prima risolvere l'ambiguità, devo esplicitare la funzione a cui faccio riferimento.

```
Studente :: fun()
```

CLASSI ASTRATTE

Con il polimorfismo non posso definire **cosa deve fare** il metodo draw(), devo però definire comunque il metodo perché altrimenti non posso usare il polimorfismo.

Le classi astratte sono classi definite a metà. Alcuni metodi hanno un corpo non definito..

Se vi sono una o più funzioni dette PURE la classe è astratta.
VIRTUALI PURE: si ottiene scrivendo fn = 0; è un modo per dire che la funzione non è definita.

```
class shape {
public:
      virtual void draw( ) = 0;          //funzione virtuale pura
      virtual void move( ) = 0;
      virtual void hide( ) = 0;
      point center;
};

class rectangle : public shape {
private:
      float lenght, width;          //dati specifici per "rectangle"
public:
      void draw( ) = {...};         //implementazione per la funzione virtuale derivata
      void move(...) =  {...};
      void hide( ) =  {...};
};
```

La dichiarazione forza tutte le classi figlie ad avere un implementazione del metodo oppure a lasciarlo astratto (ed in questo secondo caso anche la classe derivata è astratta).

STRUTTURA DI UNA CLASSE

Vediamo la struttura di una classe in C++ e le regole di visibilità.

```
class C {
public:
      //accessibili all'esterno
protected:
      //accessibili solo ai membri delle classi amiche e di quelle derivate
private:
      // accessibili solo ai membri delle classi amiche
};
```

`public:`
Sono le parti visibili globalmente, in genere usate per i metodi che il client deve poter vedere.

`private:`
Sono visibili in maniera privata, cioè accessibili all'interno della classe e a tutti gli oggetti di quel tipo. La parte private è usata per i metodi utili solo alla classe.
In genere viene usata per la struttura dati.

`protected:`
Consente l'accesso a tutte le classi che ereditano da quella che stiamo considerando. Usata per metodi e strutture dati che si vuole mantenere privato rispetto ai client ma pubblico rispetto ad estensioni della classe.

NOTA

Nella dichiarazione di una nuova classe in C++ dico come eredito da una certa classe, in particolare se ciò che eredito è disponibile pubblicamente.

```
class Counting_Stack : public stack {
```

Con la dichiarazione in alto, ciò che era public in Stack, lo è anche in Counting_Stack

```
class Counting_Stack : private stack {
```

In Counting_Stack è definito size come pubblico ed è visibile all'esterno, tutto quello che eredita da Stack è invece privato. Se faccio

```
Counting_Stack cs = new Counting_Stack();
cs.push();
```

da errore perché è ereditato in maniera privata e posso usarlo **solo** all'interno della classe.

Eredito **solo l'implementazione della classe** e non **l'interfaccia**.

DESIGN OO PURO

- La decomposizione è fatta SOLO su classi
- Una classe è sia un modulo che un tipo
- Le classi possono essere generiche
- Le classi possono essere astratte
- Relazioni tra i moduli (classi)
  - USES
  - INHERITS

IL PROBLEMA DELLE DIMENSIONI NEI TIPI PRIVATI

Se stack è un ADT che esporta un tipo stack

`s: stack;` //quanta memoria dobbiamo allocare?

DIMENSIONE DI UN ADT

Il compilatore deve sapere la dimensione lato client anche se non fa parte della specifica.

- C++ e ADA richiedono una rappresentazione dei tipi privati nella specifica (anche per la parte privata) così che quando compilo il programma devo anche avere la rappresentazione della classe (so quanto è grande).

- Java usa puntatori agli ADT che sono di una dimensione fissata. La separazione tra public e private avviene quindi anche a run-time.

IL CASO DI JAVA

Tutti gli oggetti sono implicitamente accessibili tramite puntatore

ECCEZIONI NEL C++

Nel C++ non esiste di fatto un vero e proprio gestore delle eccezioni, bensì un meccanismo che consente ad una funzione di tornare un oggetto diverso da quello definito nel caso si verifichi un errore.
Se voglio fare un inserimento nella pila, normalmente devo tornare la pila con l'elemento in più: se si verifica l'errore che la pila è piena, questo non è possibile, e dovrò tornare un oggetto diverso, per esempio un messaggio che dica perché non si è riusciti ad inserire l'elemento.
Abbiamo visto che in ADA, con il sollevamento di un'eccezione termina il sottoprogramma interessato: in un linguaggio ad oggetti, dire che un oggetto terminadi esistere, non è una cosa semplice, perché devono essere richiamate tutte le classi distruttrici relative a quell'oggetto. Inoltre, se l'oggetto è stato definito usando altri oggetti, si dovrebbero richiamare le classi distruttrici di tali oggetti, ma bisogno vedere se questi sono stati creati, altrimenti non avrebbe senso distruggerli.
Per questo motivo il gestore delle eccezioni non è implementato in modo completo nel C++, anzi quello che c'è non è nemmeno standard.
In C++ è possibile gestire, sollevare e generare le eccezioni.
Attivazione di una eccezione:

throw <nome_eccezione>

Catch è una funzione di libreria che ha come parametro il nome dell'eccezione, e ad essa si associano le istruzioni da eseguire. Esempio:

```
throw overflow

catch (overflow)
{...                                    gestione dell'overflow
}

try <insieme di istruzioni>
```

Si occupa di cercare l'eccezione. Ad essa viene associato un blocco di istruzioni che devono essere testate per errori, ovvero nel blocco si deve attivare la gestione delle eccezioni.

Esempio:

```
void main ()
{try
  calcola (...);
}
catch (overflow)
{...
}
```

Dove la funzione calcola contiene il throw.

```
void calcola (...)
{...
  if (a>max and b > max)
        throw overflow
}
```

C++ e Java

La gestione delle eccezioni in C++ è molto simile a quella di Java, tuttavia in C++:

- Non c'è obbligo di dichiarare nell'interfaccia le eccezioni sollevate da una funzione
  - E' possibile farlo, es. void foo() throw (Ex1, Ex2);
  - E' possibile impedire la propagazione di eccezioni dichiarando una lista vuota, es. void foo() throw();
  - Se non viene dichiarata una lista di eccezioni, tutte le eccezioni vengono propagate
- Se una funzione termina sollevando un'eccezione che non è dichiarata, non c'è propagazione e la funzione (ridefinibile) di sistema unexpected() viene chiamata, che termina il programma
- Quando un'eccezione viene propagata e non viene trovato uno handler, la funzione (ridefinibile) di sistema terminate() viene chiamata, che termina il programma
- Non esiste un blocco finally
- Non esiste distinzione tra eccezioni run-time e non