# *Linguaggi di Programmazione*

*Ivano Salvo*

## OOP7: Eccezioni in C++
## Miscellanea

Corso di Laurea in **Matematica**


SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **11**, 15 dicembre 2020

# *Lezione 11a*

# *Eccezioni*

# *Eccezioni*

I programmi si trovano spesso in **situazioni anomale**.

- ricevono **dato sbagliato** o incoerente;
- tentano di stabilire una connessione remota che fallisce o ritarda (time-out) o altra **interazione con l'ambiente**;
- Errori di programma, segnalati dal run-time system:
  - divisione per zero;
  - puntatore fuori dallo spazio di indirizzamento;
  - metodo non trovato.

# *Problemi e soluzioni "artigianali"*

Molti programmi **non possono semplicemente arrestarsi** (ad esempio, software di controllo di impianti)

Gestire un'eccezione "manualmente" (ad esempio riservando **valori speciali** tra i valori di ritorno di una funzione) è generalmente poco "pratico".

È generalmente ritenuto **fonte di complicazioni ed errori** mescolare il codice "normale" e quello "eccezionale".

**L'eccezione spesso deve essere rimbalzata** verso strati di codice "lontani" che vanno a gestire la situazione "eccezionale" e cercano di ripristinare uno stato consistente della computazione.

# *I tre momenti di un'eccezione*

Nei Linguaggi di Programmazione moderni esistono costrutti linguistici per gestire in modo "maturo" le eccezioni.

- **Sollevamento** (**throwing**): avviene un evento anomalo che causa la segnalazione che una situazione anomala si è verificata;
- **Cattura** (**catching**): un pezzo di codice (**handler**) cattura il segnale e mette in esecuzione del codice che la tratti;
- **Trattamento** (**handling**):
  1. informa l'utente dell'evento dell'evento anomalo;
  2. cerca di riparare;
  3. ripristina (se possibile) uno stato consistente con la computazione in atto (**resumption**) o **termina** il programma.

# *Sollevamento delle eccezioni*

Un'eccezione può essere sollevata dal **run-time system** (come ad esempio Segmentation-Fault oppure Division-by-Zero) oppure può essere **sollevata esplicitamente** dal programmatore, con il comando C++ (simile in Java):

```
throw myError("something wrong happended");
```

In ogni caso, le eccezioni sono oggetti: `myError`, in particolare, è il nome di una classe. La **stringa viene passata al costruttore**.

Usualmente, ogni diverso tipo di errore viene assegnato a una classe diversa.

**Esempio**: nel progetto scacchi, una possibilità alternativa è quella di definire il metodo `puoiMuovere` di tipo `bool` e quando la mossa è errata sollevare diversi tipi di eccezioni, come `casellaVuota`, `turnoErrato`, `reSottoScacco` etc., una per ogni cosa che può andare storta.

# Cattura delle eccezioni

Per catturare un'eccezione, occorre racchiudere il codice che potenzialmente può generare un'eccezione dentro un blocco del tipo **`try-catch`**.

Il codice che gestisce le eccezioni va scritto nell'opportuno livello di astrazione, **senza interferire** con la cosiddetta "business logic" dell'applicazione.

Se l'eccezione sfugge a tutti i catch, il programma si blocca.

```
try {
   puoiMuovere(da, a);
      /* più funzioni potrebbero generare le
       * stesse eccezioni */
   }  catch (casellaVuota e1){/*code*/}
      catch (pezzoNonMosso e2){/*code*/}
      catch (turnoErrato e3){/*code*/}
         …
      catch (otherErrors e4){/*code*/}
      catch (...){…}
        /* questo cattura ogni eccezione */
```

# *Matching delle eccezioni*

Le clausole catch vengono esaminate **in ordine testuale**.

La **prima** clausola per cui il **tipo** dichiarato è **sottotipo dell'eccezione sollevata** viene attivata.

Quindi: nel caso in cui le eccezioni (che sono classi/tipi) siano in **gerarchia**, occorre sempre **mettere prima le più specifiche** (**sottoclassi**) che altrimenti non verrebbero mai attivate.

C'è una differenza rispetto all'**overloading** dei metodi in cui viene scelto il **metodo più specifico** (potrebbe accadere che ci sono metodi con parametri di classi/sottoclassi).

# *Eccezioni e flusso di esecuzione*

Le eccezioni **modificano il flusso dell'esecuzione** di un programma.

Il blocco in cui viene sollevata un'eccezione smette di eseguire: se non è di un blocco `try-catch` che la tratta, la **funzione/metodo smette di eseguire**.

Il **controllo passa al chiamante**: se la chiamata non è dentro un blocco `try-catch` che la tratta il controllo passa al chiamante, e così via fino al caso in cui **smette di eseguire il** `main`.

In questo caso, si dice che la funzione è **sfuggita** (**escape**) e **tutto il programma si blocca**.

# Specifica di un'eccezione

Le eccezioni andrebbero **specificate nel prototipo** di una funzione, per evitare che chi usa una funzione debba spulciare il codice per vedere quali eccezioni possono essere sollevate.

```
bool puoiMuovere(Casella da, Casella a)
   throw (casellaVuota, casellaOccupata,
          turnoErrato, …);
/* occorrerebbe listare *tutte* le eccezioni */

void f();
/* significherebbe che f può sollevare
   qualsiasi tipo di eccezione */

void f() throw ();
/* significa che f non solleva *nessuna*
   eccezione */
```

# *Eccezioni "inaspettate"*

Si può definire una funzione che viene messa in esecuzione quando viene sollevata **un'eccezione non dichiarata** e non catturata.

In C++, il default per unespected è semplicemente terminare il programma. Il programmatore può scrivere una funzione che viene messa in esecuzione nel caso sia sollevata una eccezione che **non appartiene all'insieme di eccezioni dichiarate**.

```
set_unespected(myUnespected);

void myUnespected(){
   printf("unespected exception thrown\n");
   exit(1);
}
```

# *Masking / Reflecting*

A volte l'handler si limita a rimbalzare la stessa eccezione (**reflecting**), oppure a lanciarne un'altra (**masking**).

Questo può sembrare sciocco, ma le eccezioni possono essere diverse a seconda del livello di astrazione da cui si guardano.

Un errore di basso livello (tipo un accesso sbagliato in memoria) può essere visto a un livello più alto come un errore logico (ad esempio un dato aspettato e non presente)

```
/* in una funzione */
if i>n then throw ArrayOutOfBounds;

/* nell'handler associato */
catch(ArrayOutOfBound e){
   throw notFound();
   }
```

# *Overhead: non abusare*

Il meccanismo delle eccezioni è molto potente e potrebbe essere usato anche per programmare.

Tuttavia è meglio **non abusarne**: il sollevamento/cattura di un'eccezione è oneroso: occorre creare un oggetto, sospendere l'esecuzione di una funzione (con deallocazione della memoria occupata etc.).

**Non usare** eccezioni **per il normale flusso di controllo**, per ordinarie condizioni di errori, ma solo per situazioni davvero inaspettate:

# *Lezione 11b*

# *Ereditarietà Multipla*

# *Ereditarietà multipla: feature o bug?*

Nei linguaggi a oggetti puri (Smalltalk, Java etc.) ogni classe ha **una e una sola classe** madre.

La **gerarchia** delle classi **è un albero radicato**: in particolare esiste una classe alla radice della gerarchia, chiamata **`Object`** (da cui si eredita implicitamente quando non si specifica nessuna superclasse).

Questa classe ha particolari funzioni: **implementa alcuni metodi comuni** a tutte le classi (uguaglianza, clonazione, etc.) e può essere usata per **"simulare" il polimorfismo**.

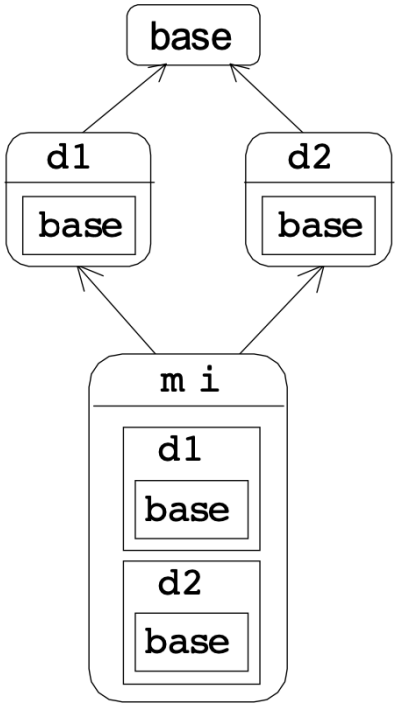
Originariamente in Java e Smalltalk, le classi contenitore erano definite di Object in modo da poter contenere per sottotipaggio qualsiasi tipo di oggetto (salvo poi dover usare il downcast per poterlo usare dopo averlo estratto).

# *Ereditarietà multipla: feature o bug?*

In **C++**, la gerarchia delle classi **non ha una root class**.

È possibile che una classe **possa ereditare da più classi**.

Questa possibilità genera una serie di problemi: il primo sono i cosidetti **diamanti**: ci sono duplicazioni di sottooggetti provenienti dalla stessa classe base e se ci sono metodi con lo stesso nome non è ben chiaro quale metodo si eredita (occorre specificare).



# Esempio: Regina

Nel progetto scacchi, ho definito un tipo di pezzo, Regina che eredita da Torre e Alfiere (l'idea è che il movimento della Regina è il movimento combinato degli altri due pezzi).

Osservare che non si tratta di una relazione IS-A, quanto piuttosto di una **specializzazione funzionale**.

Il costruttore costruisce due super-oggetti: la posizione è replicata.

```
class Regina: public Torre, Alfiere {
/* alternativa a Donna che eredita
 * sia da Torre che da Alfiere.
*/
Regina(Colore c, int x, int y)
   : Alfiere(c, x, y), Torre(c,x,y){
}
```

# Esempio: Regina

L'invocazione di super-metodi non dà problemi perché in C++ devo specificare le super-classi (quando voglio evitare la ricorsione).

Globalmente, non è una grande idea ☺

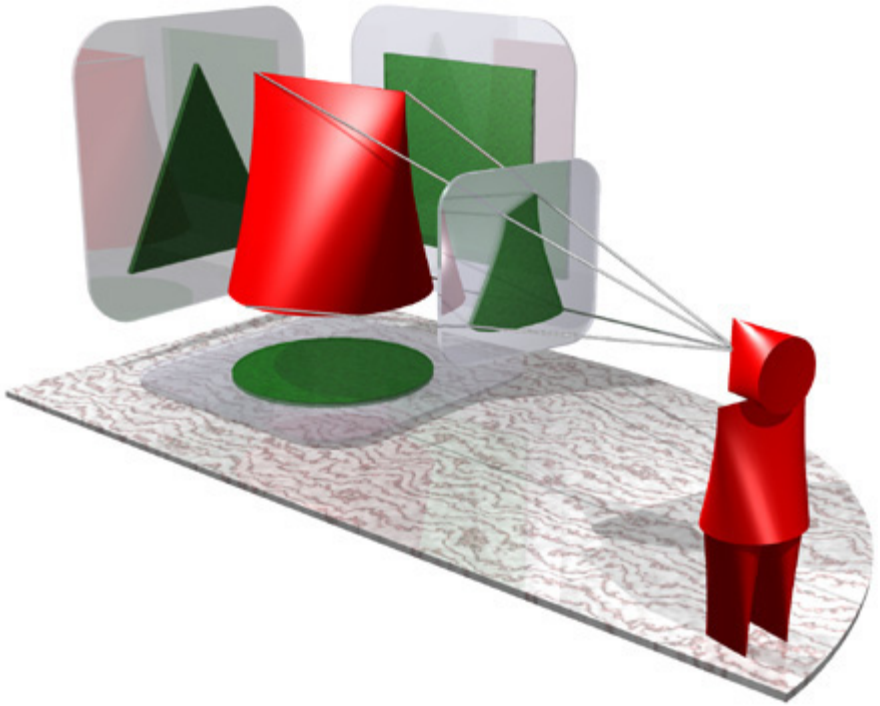
```
int Regina::puoiMuovere(Casella da, Casella a,
                        Scacchiera s){
   int res = Alfiere::puoiMuovere(da,a,s);
   if (res!=LEGALE)
            res=Torre::puoiMuovere(da,a,s);
   return res;
}
```

# *Ereditarietà Multipla ed Interfacce*

L'ereditarietà multipla sembra servire più per ragioni di sottotipaggio: voglio che un oggetto possa essere utilizzato con le sue diverse nature.

Pensate alle classi di Haskell: ci sono funzioni che richiedono un tipo parametro essere sia `Num` che `Ord`.

In Java questo problema viene risolto con le **interfacce** che sono solo prototipi, e rappresentano diverse "viste" della stessa classe.



# *Lezione 11c*

# *Run-Time Type Identification (Reflection)*

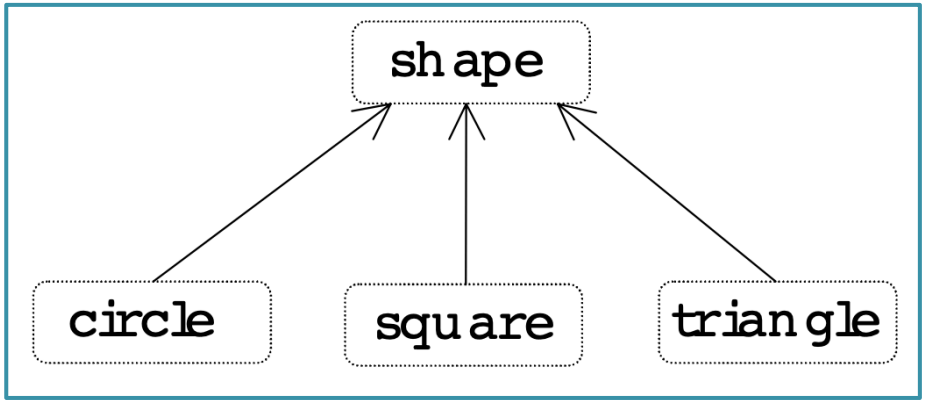
# *Run-time type identification*

Run-time type identification permette di **scoprire il tipo esatto** di oggetto quando avete un pointer o una reference (di tipo non necessariamente uguale, usualmente maggiore).

Usualmente, **conoscere il tipo esatto** di un oggetto **non è necessario** e il meccanismo dei **metodi virtual** dovrebbe far funzionare "magicamente" le cose, senza questa informazione.

Qualche volta, può essere utile ispezionare a tempo di esecuzione il tipo degli oggetti.

**Esempio**: nella gerarchia in figura, ho un insieme di `shape`, ma voglio fare un'operazione su tutti i `triangle` (colorarli di rosa)

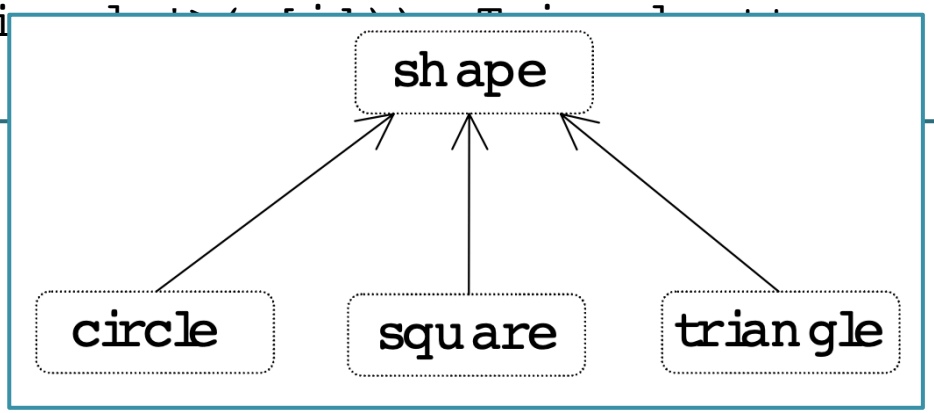


# Uso: ispezionare il tipo in collezioni

Avendo un vector di Shape, posso ad esempio contare quanti cerchi, quadrati e triangoli ci sono.

La pseudo funzione `dynamic_cast<T*>(obj)` ritorna il pointer di `obj` se il suo **tipo dinamico** è `T*`, e `NULL` altrimenti.

```
vector<Shape*> v;
…
for(int i=0; i< v.size(); i++){
   //((Circle*)v[i])->raggio; rischioso!
   v[i]->draw(); Circle* c; Square* s;
   if(c=dynamic_cast<circle*>(v[i])) nCircles++;
   if(s=dynamic_cast<square*>(v[i])) nSquares++;
   if(dynamic_cast<tri
}
```



# *Uso: safe downcast*

Viene usato per esempio quando una classe derivata **aggiunge un metodo**. In tal caso, per invocare tale metodo senza rischi, non sapendo il vero tipo del riferimento, faccio un cosiddetto **downcast safe**.

Nell'esempio, invocare `m` su `A` **non è sicuro**, in quanto `m` è definito solo nella classe derivata `B`.

Osserviamo che l'**upcast** è **sempre safe**.

```
class A{}
class B : A {
   public:
      void m(){ printf"B:42\n");}
}

void f(A* a){
   if (dynamic_cast<B*>(a) a->m();
}
```

# *Ispezionare il tipo*

C'è un'altra forma: la pseudo-funzione `typeid()`: torna un oggetto della classe `typeinfo`, su cui posso invocare metodi come `typename()` e applicare gli operatori == e !=.

Alcune librerie, usano metodi con nomi evocativi, tipo `isA()` oppure `typeOf()` allo scopo di permettere all'utilizzatore di ispezionare la gerarchia delle classi.

**Osservazione storica**: Prima dei generics, quando si usavano collezioni di Object in Java, il **safe downcast** era **sempre necessario** dopo aver estratto un oggetto da una collezione.

```
void f(A* a){
   B* b = new B();
   if (typeid(a)==typeid(b)) a->m();
   printf(typeid(a).typename());
}
```

# *Lezione 11*

*That's all Folks…*

*Grazie per l'attenzione…*

*…Domande?*