OO: Type Checking

ESERCIZIO 1

```
class a {
     int a1;

     a2 f(a3 p) {…}
}
class b extends a {
     int b1;
     b2 f(b3 p) {…}
}
1. a x = new a; b y = new b;
2. h1 = x. f(…);
3. x = y;
4. h2 = x. f(…);
5. y.b1 = x.a1;
6. x.a1 = 0;
7. ...
8. y = x;
9. ....
10. a x = new b;
11. h3 = x. f(…);
12. x.b1 = 0;
```

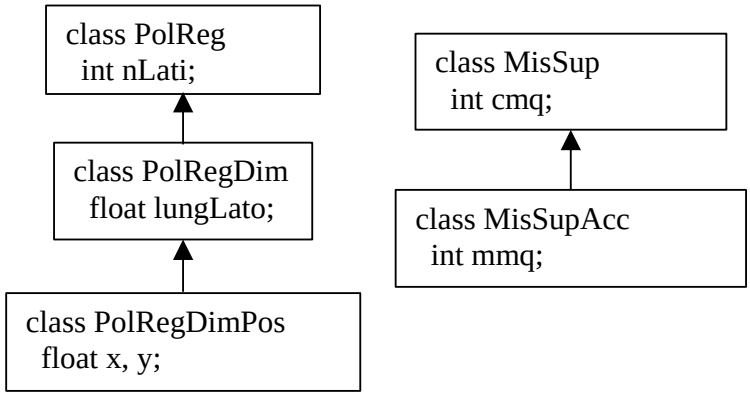
- Linguaggio *object-oriented* fortemente tipizzato in cui gli oggetti sono dinamicamente allocati nello *heap* e valgono le regole di polimorfismo e *binding* dinamico.
- Ciò che non appare dichiarato qui, come le classi a2, a3, b2, b3 e le variabili h1, h2, h3, è dichiarato altrove ed è visibile staticamente al frammento.
- Effettuare Analisi statica: istruzioni scorrette dal punto di vista del *type checking* o corrette solo sotto certe ipotesi da fare sulle parti di programma qui non mostrate.

Soluzione:

- metodi e attributi delle classi devono essere pubblici per evitare errori in compilazione.
- La ridefinizione di **f** in **b** deve soddisfare alle regole di covarianza per i risultati e controvarianza per i parametri. Solo sotto questa ipotesi, a runtime le istruzioni 4 e 11 (durante la cui esecuzione **x** risulta di tipo dinamico **b**) non generano errori.
- Le variabili h1, h2 e h3 devono essere di tipo a2 o di un super-tipo di a2.
- L'istruzione 8 è scorretta: non si può assegnare un oggetto di un tipo a una variabile che fa riferimento a un suo sottotipo.
- L'istruzione 10 è problematica. Se si trova nello stesso "*scope*" della precedente dichiarazione di **a**, questa è una doppia dichiarazione (e quindi un errore).
- L'istruzione 12 è scorretta, in quanto il *type checking* viene fatto in base al tipo statico di x (che è **a**). La classe **a** non ha un attributo pubblico b1.

ESERCIZIO 2

In un ipotetico linguaggio orientato agli oggetti si definiscono le tre classi **PolReg**, **PolRegDim** e **PolRegDimPos**, che intendono rappresentare tre nozioni di poligono regolare, via via più precise aggiungendo attributi relativi alla dimensione del lato e alla posizione (coordinate cartesiane) del centro. Similmente le due classi **MisSup**, **MisSupAcc** danno rispettivamente una misura di superficie in centimetri quadrati e una più accurata, aggiungendo un attributo per i millimetri quadrati.



Si consideri ora la classe *Geometria*, con un metodo pubblico *supPol* per calcolare la superficie di un poligono regolare:

```
class Geometria  { ...
   public  MisSup supPol(PolRegDim pol){...}
   ... }
```

e si immagini di voler introdurre una classe erede, da essa derivata, che ridefinisca il metodo *supPol*, nei seguenti quattro modi

```
class GeometriaDerivata1 inherits Geometria {...
   public  MisSupAcc supPol(PolRegDimPos pol){...}
   ... }
class GeometriaDerivata2 inherits Geometria {...
   public  MisSupAcc supPol(PolReg pol){...}
   ... }
class GeometriaDerivata3 inherits Geometria {...
   public  MisSup supPol(PolReg pol){...}
   ... }
class GeometriaDerivata4 inherits Geometria {...
   public  MisSup supPol(PolRegDim pol){...}
   ... }
```

Si indichi, scrivendo SI o NO nella seguente tabella, se ognuna delle classi derivate è corretta per quanto riguarda la ridefinizione del metodo, secondo il principio di sostituibilità, o secondo le regole dei linguaggi C++ e Java, o secondo le regole del linguaggio Ada.

	SOSTITUIBILITÀ	**C++ e Java**	**Ada**
GeometriaDerivata1	***NO***	***NO***	***SI***
GeometriaDerivata2	***SI***	***NO***	***NO***
GeometriaDerivata3	***SI***	***NO***	***NO***
GeometriaDerivata4	***SI***	***SI***	***SI***

Esercizio 3

Sia **T** una classe e **ST** una sua sottoclasse. In un ipotetico linguaggio L le variabili siano solo di tipo automatico, allocate nello stack. Si supponga anche che L consenta assegnamenti polimorfi. Ad esempio:

```
T a; ST b;
. . .
a = b;
```

Spiegare quali vincoli L dovrà imporre sulle definizioni di sottoclasse affinché possano coesistere polimorfismo e allocazione sullo stack.

Soluzione:

Il compilatore allocherà spazio per la variabile *a* in base al suo tipo statico (**T**). Gli oggetti di tipo dinamico **ST** che potranno essere assegnati ad *a* non potranno occupare piu` memoria di quella allocata dal compilatore.

Pertanto una sottoclasse non deve poter aggiungere nuovi attributi (variabili). Potrà solo aggiungere nuovi metodi e/o ridefinirne alcuni.

Variable Types:
Java: Built-in, References (only to the heap allocated memory)
C++: Automatic, Pointer, References (= const pointer that are automatically dereferenced)
Parameter passing:
Java:
Tipi base passati per copia
Oggetti per riferimento
C++:
Oggetti e puntatori passati per copia
Passaggio per riferimento tramite *reference*

Differenza semantica dell'operazione di Assegnamento: "=" in C++ e Java (1/2).

```
#include <iostream>
using namespace std;
class p { // C++
  int a;  // default: private
public:
  p(int x) { a = x; }
  int val () {return a;}
  void set(int x) {a = x; }
  ~p() { cout << "muoio: " << this << endl; }
  void praddr() {cout << this;} // mostra l'indirizzo
};
```

Oggetti sullo stack in C++

```
void main () {
p x(3);
p y(5);
p z = x;

x.praddr(); cout << "= x:" << x.val() << endl;
y.praddr(); cout << "= y:" << y.val() << endl;
z.praddr(); cout << "= z:" << z.val() << endl;

z.set(66);
x.set(11);

x.praddr(); cout << "= x:" << x.val() << endl;
y.praddr(); cout << "= y:" << y.val() << endl;
z.praddr(); cout << "= z:" << z.val() << endl;

y = z;
y.set(0);

x.praddr(); cout << "= x:" << x.val() << endl;
y.praddr(); cout << "= y:" << y.val() << endl;
z.praddr(); cout << "= z:" << z.val() << endl;

//Quando vengono distrutti gli oggetti ???
}
```

OUTPUT:

0x22feb0= x:3
0x22fea0= y:5
0x22fe90= z:3

0x22feb0= x:11
0x22fea0= y:5
0x22fe90= z:66

0x22feb0= x:11
0x22fea0= y:0
0x22fe90= z:66

muoio: 0x22fe90
muoio: 0x22fea0
muoio: 0x22feb0

Oggetti sullo heap in C++

```
void main () {
p* x = new p(3);
p* y = new p(5);
p* z = x;

x->praddr(); cout << "= x:" << x->val() << endl;
y->praddr(); cout << "= y:" << y->val() << endl;
z->praddr(); cout << "= z:" << z->val() << endl;

z->set(66);
x->set(11);

x->praddr(); cout << "= x:" << x->val() << endl;
y->praddr(); cout << "= y:" << y->val() << endl;
z->praddr(); cout << "= z:" << z->val() << endl;


delete y; // dopo y diviene irraggiungibile

y = z;
y->set(0);


x->praddr(); cout << "= x:" << x->val() << endl;
y->praddr(); cout << "= y:" << y->val() << endl;
z->praddr(); cout << "= z:" << z->val() << endl;


delete z;

} // Quando viene liberato 0xa010328 ???
```

OUTPUT:

0xa010318= x:3
0xa010328= y:5
0xa010318= z:3

0xa010318= x:11
0xa010328= y:5
0xa010318= z:11

muoio: 0xa010328

0xa010318= x:0
0xa010318= y:0
0xa010318= z:0

muoio: 0xa010318

Riferimenti in C++

OUTPUT:

```
void main () {
p& x = *(new p(3));
p& y = *(new p(5));
p& z = x;

x.praddr(); cout << "= x:" << x.val() << endl;
y.praddr(); cout << "= y:" << y.val() << endl;
z.praddr(); cout << "= z:" << z.val() << endl;
```

0xa010318= x:3
0xa010328= y:5
0xa010318= z:3

```
z.set(66);
x.set(11);

x.praddr(); cout << "= x:" << x.val() << endl;
y.praddr(); cout << "= y:" << y.val() << endl;
z.praddr(); cout << "= z:" << z.val() << endl;
```

0xa010318=x:11
0xa010328= y:5
0xa010318= z:11

```
y = z;
y.set(0);

x.praddr(); cout << "= x:" << x.val() << endl;
y.praddr(); cout << "= y:" << y.val() << endl;
z.praddr(); cout << "= z:" << z.val() << endl;
```

0xa010318=x:11
0xa010328= y:0
0xa010318= z:11

```
delete &z; delete &y

} // Quando vengono liberate 0xa010318 e 0xa010328 ???
```

muoio: 0xa010318
muoio: 0xa010328

JAVA

```
class p { // Java
    private int a;

    p(int x) { a = x; }       // default: public
    int val () {return a;}
    void set(int x) {a = x; }
};
```

```
class copia {
public static void main (String argv[]) {
p x = new p(3);
p y = new p(5);
p z = x;

System.out.println("x:"+ x.val());
System.out.println("y:"+ y.val());
System.out.println("z:"+ z.val());

z.set(66);
x.set(11);

System.out.println("x:"+ x.val());
System.out.println("y:"+ y.val());
System.out.println("z:"+ z.val());

y = z;
y.set(0);

System.out.println("x:"+ x.val());
System.out.println("y:"+ y.val());
System.out.println("z:"+ z.val());

} //Quando vengono distrutti gli oggetti ???
```

OUTPUT:

x:3
y:5
z:3

x:11
y:5
z:11

x:0
y:0
z:0

Ci pensa … il garbage collector

Differenza semantica dell'operazione di Assegnamento: "=" in C++ e Java (2/2).

VERSIONE C++

In questo esempio utilizziamo un tipo di dato astratto Foo che contiene un puntatore (data) ad una variabile di tipo carattere e un metodo per assegnare il valore a data. Nel main vengono dichiarate due variabili di tipo Foo x e y e l'una viene assegnata all'altra. Se in seguito proviamo a modificare il valore di data di x, ci accorgiamo che anche il valore di y è cambiato. Questo accade perché il costruttore di copia di default implementa un meccanismo di copia membro a membro.

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

class Foo {
private:
  char *data;
public:
  Foo() {
    data = new char;
  }

  void set(char el) {
    *data = el;
  }

  void print() {
    cout << *data << " ";
    cout << endl;
  }

};

int main() {
  Foo a;
  a.set('x');

  Foo b = a;
  a.print();
  b.print();

  a.set('y');
  a.print();
  b.print();
}
```

Per ottenere il comportamento desiderato è necessario implementare due metodi: il costruttore di copia e un metodo particolare che riguarda l'operatore di assegnamento. In entrambi i casi, l'obiettivo è di ridefinire la politica di copia tra due variabili di tipo Foo rispettivamente all'atto della dichiarazione e nel corso dell'esecuzione. In questo modo è possibile copiare il contenuto della variabile "puntata" da data e non solamente il suo indirizzo come nel caso precedente.

```
#include <iostream>
using namespace std;
class Foo {
private:
  char *data;
public:
  Foo() {
    cout << "chiamato costruttore\n";
    data = new char;
  }
  Foo(const Foo& s) {
    cout << "chiamata costruttore di copia\n";
    data = new char;
    *data = *(s.data);
  }
 /* In pratica quando viene eseguita l'istruzione var1 = var2 (con var1 e var2 entrambi di tipo Foo)
    viene eseguita questa funzione */
  Foo& operator=(const Foo& s) {
    cout << "chiamato operator=\n";
    data = new char;
    *data = *(s.data);
  }
  void set(char el) { *data = el; }
  void print() { cout << *data << " " << endl; }
};
int main() {
  Foo a;
  a.set('x');

  Foo b = a; // Foo b(a)
  a.print();
  b.print();

  a.set('y');
  a.print();
  b.print();

  b = a;
  b.set('z');
  a.print();
  b.print();
}
```

VERSIONE JAVA

Come si era già evidenziato nell'esempio sulla semantica Java, si può accedere all'istanza dell'oggetto esclusivamente tramite riferimento. Di conseguenza a e b sono esclusivamente dei riferimenti e quindi l'istruzione b = a non fa null'altro che copiare l'indirizzo contenuto in a in b ma non duplica il contenuto della cella puntata da a. Eseguendo il programma si può verificare facilmente che a e b contengono lo stesso indirizzo.

```
package javasample;
public class Foo {
    private String data;
    Foo(String testo) {
     data = new String(testo);
    }
    public void print() {
     System.out.println(data);
     }
    public void set(String testo) {
     data = data.concat(testo);
    }

    public static void main(String[] args) {
     Foo a = new Foo("Hello ");

     Foo b = a;

     a.print();

     b.print();

     a.set("world");
     a.print();
     b.print();

     System.out.println("a: " + a);
     System.out.println("b: " + b);
    }
}
```

Output:

```
Hello
Hello
Hello world
Hello world
a: javasample.Foo@3e25a5
b: javasample.Foo@3e25a5
```

Per ovviare al problema evidenziato nel programma precedente, è necessario che venga duplicato l'oggetto vero e proprio e non il riferimento. Questo è ottenuto implementando il metodo ***clone*** della classe ***Object***, specificando le operazioni che devo essere compiute per copiare in maniera corretta l'oggetto. Si noti che in questa soluzione vi sono contemporaneamente due distinte istanze di tipo DeepCopy come si può osservare dagli indirizzi stampati e dal fatto che le modifiche riferite ad *a* non vengono riportare su *b*.

```
public class DeepCopy implements Cloneable {
    private String data;
    DeepCopy(String testo) {
     data = new String(testo);
    }
    public void print() {
     System.out.println(data);
     }
    public void add(String testo) {
     data = data.concat(testo);
    }

    public Object clone() {
     DeepCopy temp;
     temp = new DeepCopy(data);
     return temp;
    }

    public static void main(String[] args) {
     DeepCopy a = new DeepCopy("Hello ");

     DeepCopy b = (DeepCopy) a.clone();
     a.print();
     b.print();

     a.add("world");
     a.print();
     b.print();

     System.out.println("a: " + a);
     System.out.println("b: " + b);
    }
}
```

Output:

Hello
Hello
Hello world
Hello
a: javasample.DeepCopy@3e25a5
b: javasample.DeepCopy @19821f

Ereditarietà

A partire dalla classe ‘persona’ definire le classi ‘studente’ e ‘professore’:

VERSIONE JAVA

```
import java.io.*; // Ereditarieta.java
import java.lang.*;
class Persona {
     String nome;
     Persona(String nome) {
          this.nome = nome;
     }
     public void print() {
          System.out.println(“Il mio nome è” + nome);
     }
}

class Studente extends Persona {
     float media;
     Studente(String nome, float media) {
          super(nome);
          this.media = media;
     }
     public void print() {
      System.out.println(“Il mio nome è “ + nome +
                         “ e la mia media è “+media);
     }
}
class Professore extends Persona {
     int pubblicazioni;
     Professore(String nome, int pubblicazioni){
          super(nome);
          this.pubblicazioni=pubblicazioni;
     }
     public void print() {
     System.out.println(“Il mio nome è      “ +
                          nome + “ e ho “ +
                          pubblicazioni + “ articoli”);
     }
}
class Ereditarieta {
     public static void main(String args[]) {
           Persona    p = new Persona(“Geometra Filini”);
           Studente   s = new Studente(“Pierino”,18);
           Professore t = new Professore(“Oronzo Cana’”,14);
           p.print();
           p = s;   p.print();
           p = t;   p.print();
     }
}
```

Output:

Il mio nome e' Geometra Filini
Il mio nome e' Pierino e la mia media e' 18.0
Il mio nome e' Oronzo Cana’ e ho 14 articoli

Nota: I metodi ridefiniti in una sottoclasse per realizzare overriding devono avere la stessa signature del metodo omonimo nella classe padre.
In Java i metodi sono ridefinibili tramite override in maniera implicita poiché il dynamic binding sui metodi è assunto per default. (in C++ è necessaria la parola chiave ***virtual***).
In tal modo è possibile realizzare il polimorfismo costringendo il sistema a eseguire un link dinamico dello stesso metodo.

VERSIONE C++

```
#include <iostream> // Ereditarietà.cpp
#include <string>
using namespace std;

class Persona {
protected:
     string nome;
public:
     Persona(string nome) {
          this->nome = nome;
     }
     virtual void print() {
          cout << "Il mio nome e'" << nome<< endl;
     }
};

class Studente : public Persona {
protected:
     float media;
public:
     Studente(string nome, float media):Persona(nome) {
          this->media = media;
     }
    void print() {
                cout << "Il mio nome e' " << nome;
                cout << " e la mia media e' " << media<< endl;
     }
};


class Professore : public Persona {
protected:
     int pubbl;
public:
     Professore(string nome, int pubbl) : Persona(nome) {
          this->pubbl=pubbl;
     }
    void print() {
      cout<<"nome: "<<nome<<"Pubbl.:"<< pubbl<<" articoli"<< endl;
     }
};
```

```
void main (int argc, char **argv) {
     Persona     *p = new Persona("Geometra Filini");
     Studente    *s = new Studente("Pierino",18);
     Professore  *t = new Professore("Oronzo Cana'",14);

      p->print();
      p = s;   p->print();
      p = t;   p->print();

}
```

ESERCIZIO: Indicare l'output del seguente programma:

```
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
        return Pet::name() + " likes to " + favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}

void main(void) {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
  // Output: "This is Fluffy" ;
  // not "Fluffy likes to sleep"
  // because the dynamic-binding is not applicable to automatic
  // variables.
}
```

ESERCIZIO RICAPITOLATIVO SUL DYNAMIC-BINDING IN C++

```
#include <iostream>
#include <string>
using namespace std;

class Instrument {
public:
  virtual void play() const {
      cout << "Instrument::play" << endl;
  }
};
class Wind : public Instrument {
public:
   void play() const { // do not need to modify the current object
         cout << "Wind::play" << endl;
   }
};

void tune(Instrument& i) { // ...
  i.play();
}

void tune_copy(Instrument i) {// ...
  i.play();
}

void main(void) {
// in C++, only pointers,reference variables (at initialization)and by-reference-parameters can be polymorphic
                                        // reference variables are of limited or no use in practice
Wind flute;       // Automatic variable
tune_ref(flute);// Parameter-passing by reference -- dynamic-binding   --- Output: Wind::play
tune_copy(flute);// Parameter-passing by value -- NO dynamic-binding --- Output: Instrument::play

Instrument piano;  // Automatic variable
piano = flute;      // NO dynamic-binding between automatic variables
piano.play();       // Output: Instrument::play

Wind& sax2 = *(new Wind); // Reference Variable
tune_ref(sax2); // Parameter-passing by reference -- dynamic-binding --- Output: Wind::play
tune_copy(sax2);// Parameter-passing by value -- NO dynamic-binding --- Output: Instrument::play

Wind& sax = *(new Wind);        // Reference Variable
Instrument& cello1 = *(new Instrument); // Reference Variable
cello1 = sax;                   // NO dynamic-binding
cello1.play();                  // Output: Instrument::play

Wind& sax1 = *(new Wind); // Reference Variable
Instrument& cello2 = sax1;// Reference Variable initialized with a
                          // copy of the un-dereferenced r-value of sax1
cello2 = sax;             // dynamic-binding
cello2.play();            // Output: Wind::play
}
```

ESERCIZIO Ereditarietà e C++.

Sia data la seguente classe Lista di interi:

```
class Lista {
public:
     // Inserisce un elemento in posizione pos se l'operazione
     // fallisce restituisce false
     bool insert(int, int pos);

     // Restituisce la lunghezza della lista
     int length();

     // Rimuove l'elemento in posizione pos se l'operazione
     // fallisce restituisce false
     bool remove(int&, int pos);

     // Restituisce l'elemento in posizione pos
     int get(int pos) const;

     // Stampa il contenuto della lista
     void print() const;
};
```

Si consideri la seguente classe astratta Pila di interi avente la seguente interfaccia:

```
class Pila {
public:
   virtual void push(int) = 0;
   virtual int pop (void) = 0;
   virtual void stampaPila() = 0;
};
```

Si realizzi una classe concreta per implementare una pila che fornisca le funzionalità di Pila, ma che utilizzi senza renderle disponibili ad un client esterno i metodi della classe Lista. Si realizzi la classe sia applicando l'ereditarietà multipla rispetto alle due classi date, sia attraverso l'ereditarietà singola dalla classe Pila e un'associazione verso la classe

Tramite ereditarietà multipla:

```
class ListPila: public Pila, private Lista {
public:
    void push(int d) {
        insert(d, 0);
    };

    int pop (void) {
       int temp;
       remove(temp,0);
       return temp;
    };
    void stampaPila(void) { print(); };
};
```

Tramite associazione:

```
class ListPila: public Pila {

private:
        Lista* lis;
public:
        ListPila(Lista* l) { lis = l; };

        void push(int d) { lis->insert(d, 0); };
        int  pop (void) {
           int temp;
           lis->remove(temp,0);
           return temp;
        };

        void stampaPila(void) { lis->print(); };
};
```

Eccezioni

- The programmer encloses in a **try** block, the code that may generate an error that will produce an exception.
- The **try** block is followed by one or more **catch** blocks.
- Each **catch** block contains an exception handler. If the exception matches the type of the parameter in one of the **catch** blocks, the code for that **catch** block is executed.
- Once an exception is thrown, control cannot return to the throw point. But control resumes with the first statement after the last **catch** block
- When an exception occurs, the exception handler receive an object and its type or value represent information from the point of the program where the exception occurred.
- Because of inheritance hierarchies it is possible that derived class object can be caught either by a handler specifying the derived-class type, or by handlers specifying the type of any base classes of that derived-class.

- Java: single inheritance from **Throwable**, **finally** for cleanup.
- C++: no static checking that **throw** statement have a corresponding throws in the function signature;

C++:

- "Catch-all" handler: **catch(...)** {}
- Rethrowing an exception: in a catch handler it is possible to have "throw;"
- It is possible to restrict the exception types thrown from a function:
    - **int** g(**double** h) **throw (**a, b,c**)** {....} // function's exception specification list
    - **int** g(**double** h) **throw** (); can throw no exception

Java:

- All the exceptions that can be thrown explicitly by a routine **must** be listed in the routine's interface:
  int g(**double** h) **throws** a, b, c {....} // function's exception specification list
  It is possible to omit the exceptions that are directly caught by the current method.
  It is possible to omit all the *unchecked* exceptions, i.e. **Error** and **RuntimeException** (e.g.: ArrayIndexOutofBoundException, NullPointerException, ClassCastException)
- the **finally** block (if present) is always executed at the end of the **try** block, whether an exception is thrown or not, unless the **try** block raises an exception that is not caught by its handlers, in which case the exception is propagated.

```
#include <iostream>
#include <string>
using namespace std;

class DivideByZeroException {
private:
  const char* message;
public:
  DivideByZeroException() :
                        message("\nAttempted to divide by 0!!!") {}
  const char* what() const { return message; }

};

double quotient(int num, int den) {
   if (den == 0) throw DivideByZeroException();
   return static_cast<double>(num) / den;
}

void main() {
 int n1, n2;
 double res ;
 cout << "\nn1 = "; cin >> n1; cout << "\nn2 = "; cin >> n2;
 try {
       res = quotient(n1,n2) ;
       cout << "\nQuotient = " << res << endl;
 }
 catch (DivideByZeroException ex) {
      cout << "\nException occurred : " << ex.what();

 }
 cout << "\nEnd of the Program!";
}
```

JAVA VERSION

```
package javasample29ott07;
import java.io.*;
import java.lang.*;
// DivideByZeroException.java
// Exception and Error are the two classes derived from Throwable, their subclasses can be thrown by the
// clauses throw, throws
class myDivideByZeroException extends Exception {
     private String message;
     public myDivideByZeroException () {
          message = "Attempted to divide by 0!";
     }
     public myDivideByZeroException (String message)  {
          this.message = message;
     }
     public String toString() { return message; }
}
// Main.java
public class Main {
   public static void main( String args[]) throws IOException {
        // E' obbligatorio riportare la lista di eccezioni
        // che potrebbero essere sollevate all'interno del metodo e che non sono
        // catturate dal metodo stesso.' In genere throws Exception va sempre bene.
        // Nel nostro caso IOException e' un'eccezione relativa alla classe RunTimeException e poteva
       // essere omessa!
          int n1, n2;
          double res;
          DataInputStream in = new DataInputStream(System.in);
          System.out.println("n1 = ");
          n1 = Integer.parseInt(in.readLine());
          //Readline può sollevareIOException
          System.out.println("n2 = ");
          n2 = Integer.parseInt(in.readLine());
          try {
               res = quotient(n1, n2) ;
               System.out.println("Quotient = "+ res) ;
          }
          catch (myDivideByZeroException ex) {
      System.out.println("Exception occurred : " + ex.toString());
          }
          finally {
                    System.out.println("Finally block");
          }
          System.out.println("End of the Program!");
    }

    static double quotient (int num, int den) throws
          myDivideByZeroException {
          // E' obbligatorio che ogni metodo che solleva esplicitamente una eccezione
          // la elenchi esplicitamente'
                if (den == 0) throw new myDivideByZeroException();
                return (double) num / den;
    }
}
```

TEMPLATE C++

Funzioni generiche

```
#include <iostream>
using namespace std;
template <class X> void myswap(X &a, X &b);
// "swap" is a name already defined in the namespace std.
void main (void) {
     int i = 10, j = 20;
     double x = 10.1, y =23.3;
     char a= 'x' , b = 'z' ;
     cout << "\n i -- j  = " << i << " " << j << endl;
     cout << "\n x -- y  = " << x << " " << y << endl;
     cout << "\n a -- b  = " << a << " " << b << endl;

     cout << "\n myswap:";
     myswap(i, j); myswap(x, y); myswap(a, b);

     cout << "\n i -- j  = " << i << " " << j << endl;
     cout << "\n x -- y  = " << x << " " << y << endl;
     cout << "\n a -- b  = " << a << " " << b << endl;
}
template <class X> void myswap(X &a, X &b) {
    X t = a;
    a = b;
    b = t;
}
```

Overload esplicito di una funzione generica:

```
#include <iostream>
using namespace std;
template <class X>
void myswap(X &a, X &b) {
    X t = a;  a = b;  b = t;
}
void myswap(int &a, int &b) {
// effettua l'override della funzione generica (la "nasconde")
    int t = a;  a = b;  b = t;
}
void main (void) {
     int i = 10, j = 20;
     double x = 10.1, y =23.3;
     char a= 'x', b = 'z';

     cout << "\n i -- j  = " << i << " " << j << endl;
     cout << "\n x -- y  = " << x << " " << y << endl;
     cout << "\n a -- b  = " << a << " " << b << endl;

     cout << "\n myswap:";
     myswap(i, j); myswap(x, y); myswap(a, b);

     cout << "\n i -- j  = " << i << " " << j << endl;
     cout << "\n x -- y  = " << x << " " << y << endl;
     cout << "\n a -- b  = " << a << " " << b << endl;
}
```

CLASSI GENERICHE

```
#include <iostream>
using namespace std;
const int SIZE = 5 ;

template <class QType>
class queue {
private:
    QType q[SIZE] ;
    int front;  // points to the current element at the head of the queue
                // which will be read at the next de-queue operation
    int rear;   // points to the current element at the tail of the queue
                // which will be written at the next en-queue operation
    int fill;  // stores the number of elements currently memorized in the queue
public:
    queue() { front = rear = fill = 0; }
    int qput(QType el) ;
    int qget(QType& el);
    int isEmpty() { return (fill == 0); }
    int isFull() { return (fill == SIZE); }
    int qlen() { return SIZE; }
};

template <class QType> int queue<QType>::qput(QType el) {
   if (( rear == front )&&( fill == SIZE )) {
     cout << "\nThe Queue is Full: the current element has not been added!";
     return (0);
   }
   q[rear] = el;
   rear = (rear + 1) % SIZE;
   fill++;
   return (1);
}

template <class QType> int queue<QType>::qget(QType& el) {
   if (( front == rear)&& (fill == 0)) {
     cout << "\nThe Queue is Empty: no element has been returned!";
     return (0);
   }
   el = q[front];
   front = (front + 1) % SIZE;
   fill--;
   return (1);
}
```

```
void main (void) {

      int i, res;

      queue<char> Qc;
     char elc;

     for (i=0; i < 5; i++) {
        elc = static_cast<char>(static_cast<int>('A')+i);
        cout << "\nenqueue Qc: " << elc;
        res = Qc.qput(elc) ;
      cout << " -- Returned " << res;
     }

     for (i=0; i < 3; i++) {
        cout << "\ndequeue Qc: ";
      if (res = Qc.qget(elc)) cout << elc;
        cout << " -- Returned " << res;
     }

    queue<double> Qd ;
    double eld;

    for (i=0; i < 5; i++) {
        eld = static_cast<double>(10.3*static_cast<double>(i));
        cout << "\nenqueue Qd: " << eld;
        res = Qd.qput(eld) ;
      cout << " -- Returned " << res;
     }

    for (i=0; i < 3; i++) {
        cout << "\ndequeue Qd: ";
       if (res = Qd.qget(eld)) cout << eld;
        cout << " -- Returned " << res;
     }

} // end main
```

TEMPLATE E OVERLOADING DEGLI OPERATORI

```
#include <iostream>
#include <string>
using namespace std;
const int SIZE = 10;


class MyIndexOutofRangeException {
private:
      string message;
public:
     MyIndexOutofRangeException() : message("\nIndex Out of Range\n"){}
     string what() { return message; }
};


template <class Atype>
class atype {
private:
    Atype a[SIZE] ;
public:
    atype();
    Atype& operator[](int i);

};

template <class Atype> atype<Atype>::atype() {
     for (int j = 0; j < SIZE; j++) a[j] = j;
}

template <class Atype> Atype& atype<Atype>::operator[](int i) {
   if ((i < 0) || (i > SIZE-1))
       throw MyIndexOutofRangeException();
   else
        return a[i];
}

void main (void) {

     int i;

     atype<int> intob ;

     try {
             cout <<"\n array di interi: " ;
             for (i=0 ; i<SIZE ; i++) { intob[i] = 2*i; }
             for (i=0 ; i<SIZE ; i++) { cout << intob[i] << " ";}
             cout << endl;
     }
     catch (MyIndexOutofRangeException h) {
          cout << h.what();
     }
```

```
	atype<double> doubleob;
	try {
		cout <<"\n array di double: " ;
		for (i=0 ; i<SIZE ; i++) { doubleob[i] = 2.5*i; }
		for (i=0 ; i<SIZE ; i++) { cout << doubleob[i] << " "; }
		cout << endl;

		doubleob[-1] = 5; // Eccezione;
	}
	catch (MyIndexOutofRangeException h) {
	     cout << h.what();
	}
}

// per Esercizio provare ad implementare un array a dimensione variabile
// con controllo degli indici; specificando la dimensione dell'array da
// dichiarare nel costruttore e allocare dinamicamente la sequenza.
```