Appunti lezione

Ripasso

Esistono contesti diversi del linguaggio di programmazione

```
#include <iostream>

//opzione -E : solo preprocessing
//opzione -S : fino ad assembler
//opzione -c : fino al codice oggetto quindi .o
//le opzioni ci dicono dove si deve fermare il compilatore

int main(){
    std::cout<<"Hello World"<<std::endl;
}
```

Namespace? Libreria standard

endl non è riconosciuto dal linguaggio

```
std::endl
```

std è la libreria

```
"Hello world"
```

E' una variabile letteraria, un array double di 14 caratteri

Variabili

A cosa servono i puntatori "opachi"?

Sono utili per fare ad esempio l'information hiding

Qual è la differenza tra dichiarazione e definizione?

```
extern int a; // dichiarazione pura di variabile globale
int b; // definizione di variabile
int c = 1;
extern int b = 1; // questa è una definizione perchè c'è l'inizi
```

Funzioni

```
void foo(int a ); // dichiarazione pura
extern void foo(int a); // dichiarazione pura

void foo(int a) {std::cout<<a;} // dichiarazione e definizione
```

Campo d’azione (scope)

1. **Scope di namespace** (incluso lo scope globale)

   Il namespace globale include il “niente”

   ```
   :: // è incluso nel namespace globale
   ```

   quindi se volessi potrei scrivere una variabile in diversi modi nel namespace globale come:

   ```
   int a;
   cout<<a<<;
   // oppure
   int a;
   cout<<::a<<; // :: indica il namespace globale
   ```

   Esempio:

   ```
   namespace N {

       void foo() {
         // errori: bar e a non sono visibili in questo punto (d
         bar(a);
       }

       int a; // definizione di a

       void bar(int n) {
         a += n; // OK: a è visibile in questo punto (dichiarata
       }

   } // namespace N
   ```

💡 **Nota**: visibilità dello scope

```
namespace N{
    int b;
}//namespace N

namespace M{
    int foo(){
        return b; // mi darà errore in quanto b non si
    }
}//namespace M
```

per correttezza bisogna cercare b nel namespace N

```
namespace M{
    int foo(){
        return N::b; // visto che b non si trova nel namespac
    }
}//namespace M
```

2. **Scope di blocco**

   Le definizioni nei blocchi sono valide fino alla fine del blocco

   Esempio:

```
for(int i=0;i<5;i++){// i è valida fino alla fine del for
    ///
    ///
}
```

3. **Scope di classe**

   I membri di una classe (tipi, dati, metodi) sono visibili all'interno della classe indipendentemente dal punto di dichiarazione

```
struct S {
  void foo() {
    bar(a); // OK: bar e a sono visibili anche se dichiarati
  }

  int a; // a può essere usata anche prima della funzione foo
  void bar(int n) { a += n; }
};
```

Nella scope di classe non esiste la nozione di prima e dopo

Nota: i membri di una classe possono essere acceduti dall'esterno

```
s.foo();   // usando l'operatore punto, se s ha tipo (riferim
ps->foo(); // usando l'operatore freccia, se ps ha tipo punta
S::foo;    // usando l'operatore di scope.
```

Nota: i membri di una classe S possono essere acceduti anche da classi che sono derivate dalla classe S

4. **Scope di funzione**

   Le etichette di destinazione delle istruzioni goto hanno scope di funzione: sono visibili in tutta la funzione che racchiude, indipendentemente dai blocchi.

```
void foo() {
  int i;
  {
  inizio: // visibile anche fuori dal blocco
    i = 1;
    while (true) {
      // .. calcoli ...
      if (condizione)
        goto fine; // fine è visibile anche se dichiarata dop
    }
  }
fine:
  if (i > 100)
    goto inizio;
```

```
  return i;
}
```

5. **Scope delle costanti di enumerazione: un caso speciale**

   Le costanti di enumerazione dichiarate secondo lo stile C++ 2003

   ```
   enum Colors {red, blue, green};
   ```

   Sono visibili anche al di fuori dello scope dichiarato

   E’ un problema

   ```
   enum Colori { rosso, blu, verde };
   enum Semaforo { verde, giallo, rosso };

   void foo() { std::cout << rosso; } // a quale rosso si riferi
   ```

   In quanto la variabile rosso può avere valori diversi

   Nel C++ 2011 sono state introdotte le “enum class”, che invece limitano lo scope come le classi, costringendo il programmatore a qualificare il nome e evitando potenziali errori

   ```
   enum class Colori { rosso, blu, verde };
   enum class Semaforo { verde, giallo, rosso };

   void foo() {
     std::cout << static_cast<int>(Colori::rosso);
   }
   ```

   💡 **Nota:** il cast è necessario perche’ le enum class impediscono anche le conversioni implicite di tipo verso gli interi.

Casi di dichiarazione di riduzione o estensione dello scope di una dichiarazione

Hiding

Quando si annidano campi d’azione, è possibili che una dichiarazione nello scope interno nasconda un’altra dichiarazione (con lo stesso nome) dello scope esterno.

```
int a = 1; // scope globale

int main() {
  std::cout << a << std::endl;   // stampa 1
  int a = 5;
  std::cout << a << std::endl;   // stampa 5
  {
    int a = 10; // la a esterna viene nascosta
    std::cout << a << std::endl; // stampa 10
  } // lo scope della a esterna riprende da questo punto
  std::cout << a << std::endl;   // stampa 5
}
```

Anche per membri ereditati da una classe

```
struct Base {
  int a;
  void foo(int);
};

struct Derived : public Base {
  double a;           // hiding del data member Base::a
  void foo(double d); // hiding del metodo Base::foo()
};

--Estensione delle visibilità di un nome
```

Estensione delle visibilità di un nome

Se un nome deve essere usato molto spesso in un posizione in cui non è visibile senza qualificazione, si può usare la dichiarazione di tipo using

```
#include <iostream> // senza darebbe errore

void foo() {
  using std::cout;
```

```
  using std::endl;
  cout << "Hello" << endl;
  cout << "Hello, again" << endl;
  cout << "Hello, again and again and again ..." << endl;
}
```

> 💡 **Nota**
> Una dichiarazione di using può rendere disponibili solo nomi che erano stati
> precedentemente dichiarati (o resi visibili) nel namespace indicato. In particolare, nel caso precedente, è comunque necessario includere l'header file iostream, altrimenti si ottiene un errore.

La cosa è invece legittima nel caso di funzioni, perché in quel caso entra in gioco il meccanismo dell'overloading. Nell'esempio seguente, la dichiarazione di using crea l'overloading per i metodi di nome foo (evitando l'hiding)

```
struct Base {
  void foo(int);
  void foo(float);
};

struct Derived : public Base {
  using Base::foo;    // rendo visibili in questo scope tutti
                      // di nome foo presenti in Base
  void foo(double d); // foo(double) va in overloading
                      // con foo(int) e foo(float)
};
```

Il “public” davanti a base è ridondante perché le struct ha i dati membri public se non si specifica, nel caso di class invece public non sarebbe ridondante

Direttive Using

Cosa ben distinta rispetto alle dichiarazioni di using sono le direttive di using (using directive)

```
void foo() {
  using namespace std;
  cout << "Hello" << endl;
  cout << "Hello, again" << endl;
  cout << "Hello, again and again and again ..." << endl;
}
```

Tempo di vita di un oggetto

Il tempo di vita di un oggetto è influenzato dal modo in cui questo viene creato

> 💡 **Nota**
>
> anche se il codice eseguibile delle funzioni è memorizzato in memoria, tecnicamente le funzioni NON sono considerate oggetti in memoria e quindi non se ne considera il lifetime

Gli oggetti in memoria sono creati:

- da una definizione (non basta una dichiarazione pura)
- da una chiamata dell'espressione new (oggetto nell'heap, senza nome)
- dalla valutazione id una espressione che crea implicitamente un nuovo oggetto (oggetto temporaneo, senza nome)

Tempo di vita di un oggetto:

1. inizia quando *termina* la sua costruzione, che è composta da due fasi distinte:
   a. allocazione delle memoria "grezza"
   b. inizializzazione della memoria (quando prevista)
2. termina quando *inizia* la sua distruzione, che è anche 'essa composta da due fasi:
   a. invocazione del distruttore (quando previsto)
   b. de allocazione della memoria "grezza"

💡 **Nota**
Un oggetto la cui costruzione non termina con successo di fatto non ha mai iniziato il suo ciclo di vita ovvero per quel oggetto non verrà eseguita la distruzione

💡 **Nota**
Durante le fasi di costruzione e distruzione non tutte le operazioni con l'oggetto sono legittime, alcune operazioni potrebbero avere un comportamento diverso da quello atteso.

Storage duration

Viene tradotta come allocazione, ma non è così.

1. **Allocazione statica**
    a. le variabili definite a namespace scope (dette globali)
    b. i dati membro di classi dichiarati usano la parola "static"
        i. sono come le variabili globali
    c. le variabili locali dichiarate usano la parola chiave "static"
        i. Vengono inizializzate solo la prima volta in cui il controllo di esecuzione incontra la corrispondente definizione.

Due variabili globali non hanno ordine di definizione

Il seguente programma stampa due stringhe, anche se il corpo della funzione main è vuoto

```
#include <iostream>

struct S {
  S() { std::cout << "costruzione" << std::endl; }
  ~S() { std::cout << "distruzione" << std::endl; }
};

S s; // allocazione globale
```

```
int main() {}
```

Il seguente programma stampa una stringa, indicando il numero totale di volte in cui la funzione foo() è stata chiamata

```
#include <iostream>

struct S {
  int counter;
  S() : counter(0) { }
  ~S() { std::cout << "counter = " << counter << std::endl; }
};

void foo() {
 static S s; // allocazione locale statica
 ++s.counter;
}

int main() {
  for (int i = 0; i < 10; ++i) {
    foo();
  }
}
```

2. **Allocazione thread local**

   Allocazione globale ma di tipo thread local

   La variabile è inizializzata prima della creazione thread e de inizializzata prima della fine del thread

3. **Allocazione automatica**

   Una variabile locale ad un blocco di funzione è dotata di allocazione automatica

```
void foo() {
  int a = 5;
  {
    int b = 7;
    std::cout << a + b;
```

```
  } // b viene distrutta automaticamente all'uscita da questo
  std::cout << a;
} // a viene distrutta automaticamente all'uscita da questo b
```

> 💡 **Nota**
>
> nel caso di funzioni ricorsive, sullo stack possono esistere contemporaneamente più istanze distinte della stessa variabile locale.

Ogni volta che la funzione viene chiamata la variabile viene creata e poi distrutta

La de allocazione è automatica

4. **Allocazione automatica di temporanei**

   L'allocazione automatica di temporanei avviene quando un oggetto viene creato per memorizzare il valore calcolato da una sotto espressione che compare all'interno di una espressione.

```
struct S {
  S(int);
  S(const S&);
  ~S() { std::cout << "distruzione"; }
};

void foo(S s);

void bar() {
  foo(S(42)); // allocazione di un temporaneo per S(42)
  std::cout << "fine";
}
```

   La de allocazione è automatica

5. **Allocazione dinamica**

   La de allocazione non è automatica

   Il risultato della new è l'indirizzo della variabile

```
int* pi = new int(42);
  // pi contiene l'indirizzo di un oggetto int di valore 42
```

Delete de alloca l’oggetto puntato da pi

Delete può creare diversi errori:

- errore “double free”

  usare un puntatore dangling; in pratica si usa un oggetto dopo che il suo lifetime è concluso.

- memory leak

  si distrugge l’unico puntatore che contiene l’indirizzo dell’oggetto allocato prima di avere effettuato delete

- accesso ad un “wild pointer”
- accesso al “null pointer”

Tipi riferimento e tipi puntatore

I riferimenti java sono diversi dai riferimenti in C

In c++ il riferimento nullo non esiste

Un puntatore è un oggetto, il cui valore (un indirizzo) si può riferire ad un altro oggetto (necessariamente diverso) quando creo un riferimento lo devo inizializzare

Invece i puntatori possono essere definiti senza inizializzazione

Una volta creato il riferimento questo si riferisce sempre a quell’oggetto

Il puntatore può cambiare l’oggetto che punta

📎 Se voglio che non venga modificato l’argomento lo passo come const.

📎 Passare per riferimento è meglio perche’ passare col puntatore possiamo avere il puntatore a NULL. Uso il puntatore per prendere argomenti opzionali.

💡 Nuovo standard: std::optional da usare per argomenti opzionali

Tipi

Tipi di caratteri

- Narrow char type

  char, unsigned char

- Wide char type

  wchat_t, char16_t, char32_t

Tipi interi standard con segno

signed char, short, int, long, long long

Tipi interi standard senza segno

unsigned char, unsigned short, unsigned int…

📎 Tutti i tipi suddetti sono detti tipi integrali
Booleani, caratteri narrow e short sono detti tipi integrali “piccoli”, in quanto *potrebbero* avere una dimensione (sizeof) inferiore a int e pertanto sono oggetti a promozione di tipo.

📗 **Tipi integrali**: booleano, carattere e intero

Tipi aritmetici: tipi integrali e a virgola mobile

Tipi definiti dall’utente: enumerazioni e classi

Tipi incorporati: puntatori e riferimenti

Tipi composti

riferimenti a lvalue --- T&
riferimenti a rvalue --- T&&
puntatori --- T*
tipi array --- T[n]
tipi funzione --- T(T1, ..., Tn)
enumerazioni e classi/struct

Tipi qualificati: il qualificatore const

📎 Nel c++ ci sono qualificatori “const” e “volatili”

L’accesso ad un oggetto attraverso una variabile il cui tipo è dotato del qualificatore const è consentito in sola lettura.

Costanti letterali

bool:
false, true

char:
'a', '3', 'Z', '\n' (ordinary character literal)
u8'a', u8'3' (UTF-8 character literal)
signed char, unsigned char:
<nessuna>
char16_t:
u'a', u'3' (prefisso case sensitive)
char32_t:
U'a', U'3' (prefisso case sensitive)
wchar_t:
L'a', L'3' (prefisso case sensitive)

short, unsigned short:
<nessuna>

int:
12345

Array e puntatori

Quando si usa un’espressione di tipo array di T, viene applicato il type decay e si ottiene il puntatore al primo elemento dell’array.

💡 lvalue:
Riferimento a sinistra dell’operatore di conversione
es: int i (i è un lvalue)

Questa conversione server per evitare copie costose.

```
//Si considerino le due variabili
  int a[100];
  int b = 5;
//L'espressione
  a[b]
//è in tutto e per tutto equivalente all'espressione
  *(a + b)
```

Siccome la somma è commutativa, lo stesso risultato lo si ottiene anche usando l'espressione

🚫 DA NON USARE: *(b+a)

Overloading

Le prossime fasi vengono ripetere per ogni singola chiamata di funzione presente nel codice sorgente:

1. individuazione delle funzioni candiate
2. selezione delle funzioni utilizzabili
3. scelta della migliore funzione utilizzabile (se esiste)

Le funzioni candidate

L'insieme delle funzioni candidate per una specifica chiamata di funzione e' un sottoinsime delle funzioni che sono state dichiarate all'interno dell'unita' di traduzione

a) hanno lo stesso nome delle funzione chiamata

b) sono visibili nel punto della chiamata

❗ Attenzione a non confondere l’overloading con l’hiding
Esempio:

```
struct S { void foo(int); };
      struct T : public S { void foo(char); };
      T t; // tipo statico e dinamico coincidono (T)
      t.foo(5);
      // la ricerca inizia nello scope di T; la funzione S
      // non va in overloading, perché viene nascosta (hid
```

❗ Attenzione ad eventuali dichiarzioni e direttive di using, che modificano la visibilità delle dichiarazioni
Esempio:

```
      nell'esempio precedente, se nella classe T fosse agg
        using S::foo;
      allora S::foo e T::foo sarebbero entrambe visibili e
      in overloading
```

❗ Attenzione all’ADL (Argument Dependent Lookup)
La regola ADL, detta anche Koenig’s lookup, stabilisce che:

a. nel caso di una chiamata di funzione NON qualificata,

b. se vi sono (uno o piu’) argomenti ”arg” aventiun tipo definito dall’utente (cioe’ hanno tipo struct/class/enum S, o riferimento a S o puntatore a S, possibilmente qualificati) e

c. il tipo sudetto e’ definito nel namespace N allora la ricerca delle funzioni candidate viene effettuata anche all’interno del namespace N.
Esempio:

```
namespace N {

struct S { };

void foo(S s);

void bar(int n);

} // namespace N

int main() {
  N::S s;
  foo(s);   // chiamata 1
  int i = 0;
  bar(i);   // chiamata 2
}
```

Per la chiamata 1, si applica la regola ADL, perché il nome foo non è qualificato e l'argomento s ha tipo struct S definito dall'utente all'interno del namespace N. Quindi il namespace N viene "aperto" rendendo visibile la dichiarazione di N::foo(S) nel punto della chiamata (e quindi rendendola candidata).

In contrasto, per la chiamata 2, NON si applica la regola ADL, perché l'argomento ha tipo int (che non è definito dall'utente) e quindi non viene

aperto nessun namespace.

💡 **Nota**

la regola ADL è quella che consente al programma "Hello, world!" di funzionare come ci si aspetta. La chiamata std::cout << "Hello, world!" corrisponde alla invocazione di funzione operator<<(std::cout, "Hello, world!"). La chiamata non è qualificata e il primo argomento ha tipo std::ostream, che è un tipo definito dall'utente all'interno del namespace std. Quindi, il namespace std viene "aperto" e tutte le funzioni di nome "operator<<" dichiarate al suo interno diventano visibili (e quindi candidate).

Selezionare, tra le candidate, delle funzioni utilizzabili

Effettuata la scelta delle funzioni candidate, occorre verificare quali di queste funzioni potrebbero essere effettivamente utilizzabili (viable) per risolvere la specifica chiamata considerata.

Per decidere se una funzione candidata e' utilizzabile, e' necessario verificare che:

A. il numero degli argomenti (nella chiamata di funzione) sia compatibile con il numero di paramentri (nella dichiarazione di funzione)

B. ogni argomento (nella chiamata di funzione) abbia un tipo compatibili con il corrispondente paramentro (nella dichiarazione di funzione)

❗ Con riferimento alla compatibilità del numero di argomenti:

- attenzione ad eventuali valori di default per i parametri
- attenzione all'argomento implicito (this) nelle chiamate di metodi non statici

Selezione della migliore utilizzabile (se esiste)

📖 Sia N il numero di funzioni utilizzabili:
se N = 0, allora ho un errore di compilazione;
se N = 1, allora l'unica utilizzabile è la migliore;
se N > 1, allora occorre classificare le funzioni utilizzabili
in base alla "qualità" delle conversioni richieste;
se la classificazione (spiegata sotto) determina una
unica vincitrice, quella funzione è la migliore utilizzabile;
altrimenti si ottiene un errore di compilazione
(chiamata ambigua).

Classificazioni:

Per ognuno degli M argomenti presenti nella chiamata, si crea una classifica delle funzioni utilizzabili. La funzione migliore (se esiste) è quella che è preferibile rispetto a tutte le altre. Per decidere se X è preferibile rispetto ad Y, si confrontano X e Y su tutte le M classifiche corrispondenti agli M argimenti

X è preferibile a Y se:

- non perde in nessuno degli M confronti (i.e., vince o pareggia);
- vince almeno uno degli M confronti.

Conversioni Implicite di tipo

Le conversioni implicite si possono suddividere in 4 categorie

Le corrispondenze “esatte”

A. match perfetto: esattamente identico al tipo di parametro

Non e’ conversione perche’ tipo di partenza e tipo di destinazione coincidono

📎 Es: (si assumono le dichiarazioni "int i; const int& r = i;")

tipo parametro	argomento
int	5
int&	i
int*	&i
const int&	r
double	5.2 // nel caso 5.2F non piu’ match perfetto

B. trasformazioni di lvalue

📎 Es: (si assumono le dichiarazioni "int a[10];" e "void foo();")

tipo parametro	argomento	
int	i	(da lvalue a rvalue)
int*	a	(decay array-puntatore)
void(*)()	foo	(decay funzione-puntatore a funzione)

C. conversione di qualificazione

Viene aggiunto il qualificatore const

📎 Es:

tipo parametro	argomento
const int&	i
const int*	&i

Le promozioni

Preservano il valore dell'argomento corretto.

A. Promozioni intere

dai tipi interi piccoli (cahr/short signed o unsigned) al tipo int (signed o unsigned)

da bool a int e' promozione (caso speciale)

B. Promozioni floating point

da float a double

C. Promozioni delle costanti di enumerazioni del C++ 2003

al piu' piccolo tipo intero (almeno int) sufficientemente grande per contenerle

Conversioni standard

Sono tutte le convesioni non definite dall'utente e non quelle esatte

❗ Da int a long non e' promozione, ma conversione standard
Da char a double: e' conversione standard
…ecc

Tra le conversioni standard da tenere in considerazione vi sono le conversioni tra riferimenti e tra puntatori.

In particolare:

```
    - la costante intera 0 e il valore nullptr (di std::nullptr
      sono le costanti puntatore nullo; esse possono essere con
      implicitamente nel tipo T* (per qualunque T);
      la costante intera 0 può essere convertita in nullptr;
    - ogni puntatore T* può essere convertito nel tipo void*,
      che corrisponde ad un puntatore ad un tipo ignoto
      (si noti che non esiste una conversione implicita che vad
      in senso inverso, da void* a T*);
    - se un classe D è derivata (direttamente o indirettamente)
      dalla classe base B, allora ogni puntatore D* può essere
      convertito implicitamente in B*; si parla di "up-cast"
      (conversione verso l'alto), perché tradizionalmente nei
      diagrammi che rappresentano le relazioni tra i tipi di da
      le classi base, che sono più astratte e quindi "leggere",
      vengono rappresentate sopra le classi derivate, che sono
      più concrete e quindi "pesanti"; una analoga conversione
      sui riferimenti consente di trasformare un D& in un B&;
      (anche in questo caso, si noti che non esiste una convers
      implicita per il "down-cast", che trasformerebbe un B* in
      o un B& in un D&).
```

Conversioni implicite definite dall’utente

A. Uso (implicito) di costruttori che possono essere invocati con un solo argomento (di tipo diverso) e non sono marcati ‘exlicit’

Esempio:

```
struct Razionale {
  explicit Razionale(int num, int den = 1); // conv. da int a
    // ...
};
```

Se non ho explicit:

```
struct Razionale {
  Razionale(int num, int den = 1); // conv. da int a Razionale
    // ...
};
```

Se per esempio ho una funzione che prende Razione (r):

```
void foo(Razionale r);
...
foo(true);
```

Se venisse passato a foo(true) true verrebe promosso a int e avremmo 1/1, con explicit questo non accadrebbe

B. Uso di operatori di convesione da tipo utente verso altro tipo

Esempio:

```
struct Razionale {
    operator double() const; // conversione da Razionale a dou
  // ...
};
```

Anche in questo caso e' prevista utilizzare la parola 'explicit' e significherebbe che questa non sarebbe una convesione che si puo' aplicare implicitamente

---

Classe concreta (Test Driven Design)

📎 Argomenti da studiare e/o ripassare (i numeri dei capitoli si riferiscono allo Stroustrup):

- classi (cap. 16)
- costruttori, distruttori e costruttori di copia (cap. 17)
- sovraccaricamento operatori (cap. 18)
- operatori speciali (cap. 19)
- classi derivate (cap. 20)

💡 I distruttori vengono invocati implicitamente molto spesso ad esempio con le operazioni:

```
r=r+r; //invocazione implicita del distruttore
r=r-r; //invocazione implicita del distruttore
...
```

L’operazione r+r crea un temporaneo per fare l’operazione, una volta finita viene invocato il distruttore in modo implicito e il temporaneo viene distrutto

Direttive d’inclusione di un header file Razionale.hh

```
#ifndef GUARDIA_Razionale_hh
#define GUARDIA_Razionale_hh 1

class Razionale{
public:
    using Int = long;
    //costruttore di default
    Razionale() = default;

    //costruttore di copia
    Razionale(const Razionale& y) = default;

    //costruttore di spostamento
    Razionale(Razionale&&) = default;

    //assegnamento di copia
    Razionale& operator=(const Razionale&) = default;

    //assegnamento per spostamento
    Razionale& operator=(Razionale&&) = default;

    //distruttore
    ~Razionale()=default;

    //explicit Razionale(Int n);
    //Razionale(Int n, Int d);
```

```
    explicit Razionale(Int n, Int d = 1);

    Razionale operator+(const Razionale& y) const;
    Razionale operator-(const Razionale& y) const;
    Razionale operator*(const Razionale& y) const;
    Razionale operator/(const Razionale& y) const;

    Razionale operator-() const;
    Razionale operator+() const { return *this; }; //implementaz

        //tipo di this?
        void foo();        // Razionale* const
        void bar() const;  // const Razionale* const

private:
    // Int num_;
    // Int den_;

};

#endif //GUARDIA_Razionale_hh
```

Nome canonico

```
using Int = long; // uso il nome Int per i long
// quindi se ho un ripensamento piu' avanti nel codice devo camb
...
Int num_; // la variabile num_ e' di tipo long
Int den_;
...
```

🚫 Evitare di fare costruttori impliciti, quindi mettere la parola “explicit” d’avanti al costruttore

```
(void) funzione();
(void) b; // un modo per evitare warning su variabili non usate
```

```
// sintassi cast C
```

Rappresentazione canonica

💡 **Invariante di classe**
Proprietà che non varia
COSA MOLTO IMPORTANTE PER LA CLASSE

La proprietà invariante serve per avere una rappresentazione coerente per l'utente

Controllo invariante

```
bool check_inv() const{
    if(num_ == 0)
        return den_ == 1; // lettura: se il numeratore è 0 il de
    if(den_ <= 0)
        return false;
    if(std::gcd(num_, den_) != 1)
        return false;
    // qui l'invariante è soddisfatta
    return true;
}
```

```
if(check_inv())
    return;
else{
    throw ... // lancio un' eccezione: faccio un'asserzione
}
```

💡 Fare un'asserzione: valutare l'espressione
Si definisce con <cassert>

Le asserzioni sono una MACRO

Con #include <cassert>

Posso fare direttamente:

```
#include <cassert>
...
assert(check_inv())
```

```
assert(den != 0) // controllo se il denominatore è diverso da ze

if(num_ == 0)
    den_ = 1;
    // qui controllo invariante
    assert(check_inv());
    return;
}
if(den_ < 0){
    num_ = -num;
    den_ = -den_;
}
Int gcd = std::gcd(num_, den_);
if(gcd != 1){
    num_/=gcd;
    den_/=gcd;
}
```

Gli assert possono essere usati per controllare le invarianti ma anche altre cose che dovrebbero essere vere.

💡 Le asserzioni interrompono brutalmente il programma nel punto di errore

💡 **Disabilitare le asserzioni**

Si definisce una macro nel preprocessore

```
g++ ...-DNDEBUG...
```

Lista inizializzazione dei dati membro

L’inizializzazione dei dati membro di una funzione e’ molto utile quando si tratta di dati di tipo particolare come stringhe tipo c

```
explicit Razionale(Int n, Int d=1)
    : num_(n), den_(d) {}
```

Usando ad esempio un tipo di dato stinga tipo c inizializzare il dato nelle grafe significia prima inizializzare mettendoci una stringa vuota e dopo questa stringa vuota verra’ sovrascritta usando un assegnamento.

Dobbiamo inoltre normalizzare la

> 💡 **Normalizzare**
>
> Significa avere una funzione privata che mette in forma normale i miei dati

---

Precondizione e Postcondizione

L’invariante di classe e’ parte delle precondizioni e postcondizioni dei metodi della classe.

Questo formano il “contratto” che lo sviluppatore stabilisce con l’utilizzatore della classe.

```
...
void foo(){
    assert(check_inv()) // precondizione
    ...
    ...
    assert(check_inv()) // postcondizione
}
```

precondizioni ⇒ postcondizioni , si legge cosi:

Se le precondizioni (a carico dell’utente) sono vere allora (l’implementatore) mi impegno a far in modo che anche le postcondizioni siano vere.

💡 **Nota**
Se le precondizioni non sono valide (sono false) allora l'implicazione del contratto e' vera a prescindere dalla validita' o meno delle postcondizioni. Ovvero se l'utente non soddisfa una precondizione l'implementatore non ha alcun obbligo.

Contratti

Contratti Narrow (stretti)

Restringiamo l'insieme dei valori che sono ammessi ad invocare il metodo o la mia funzione.

L'implementatore si impegna a fornire la funzionalita' solo quando ha senso farlo, cioe' quando i valori forniti in input sono legittimi; l'onere di verificare tale legittimita' e' lasciato all'utilizzatore.

💡 I contratti narrow sono molto comuni in c++ in quanto consentono di fare meno test a livello di implementazione, quindi piu' efficienza

Contratti Wide (ampi)

Nei contratti wide l'onere di verificare la legittimita' delle invocazioni ricade sull'implementatore.

Scegliere un contratto wide equivale a spostare alcuni elementi del contratto dal lato della precondizione al lato della postcondizione.

💡 I contratti wide portano a codice inefficente e lento

Gestione delle risorse e Exception Safety

📎 Argomenti da studiare e/o ripassare (i numeri dei capitoli si riferiscono allo Stroustrup):

- capitolo 13: Gestione delle eccezioni
- capitolo 17: Costruzione, pulizia, copia, spostamento

💡 **Risorsa**

Qualche entita' disponibile in quntita' limitata, se si esaurisce posso compromettere la funzionalita' del software

L'interazione del software con le risorse deve avvenire secondo uno schema predefinito, suddiviso in tre fasi ordinate temporalmente:

1. acquisizione della risorsa
2. uso della risorsa
3. restituzione (rilascio) della risorsa

In particolare:

- la risorsa deve essere acquisita prima di essere usata (o rilasciata);
- al termine del suo utilizzo la risorsa deve essere rilasciata;
- non e' lecito usare una risorsa dopo averla rilasciata (se occorre, si puo' ripartire dalla fase 1)

Esempi di risorse:

- la memoria ad allocazione dinamica
  - viene acquisita tramite l'uso dell'espressione **new**, utilizzata per leggere e scrivere valori e infine rilasciata mediante l'uso della **delete**;
  - il mancato rilascio generea "memory leak", e in casi particolari puo' portare all'esaurimento della memoria disponibile.
- i (descrittori dei) file del file system
- i lock per l'accesso a risorse condivise;
- le connessioni di rete a server (e.g., sessioni DBMS);
- ecc

Esempio banale:

```
void foo(){
    int* pi=new int(42);
    do_the_job(pi);
    delete pi;
}
```

Nel caso in cui nella funzione `do_the_job()` ci fosse una gestione di un'eccezione, ma non fosse gestita internamete e fosse propagata all'esterno, si uscirá dalla funzione `do_the_job()` in modalita' eccezionale si salterá l'istruzione `delete pi` , quindi verrá cancellato il puntatore pi ma non la risorsa e quindi avremo **memory leak.**

💡 Quando e' legittimo che il codice non sia exception safe:

1. la risorsa in questione é poco importante
2. la correttezza del software (nei casi in cui si presenta un comportamento eccezionale) é di scarso interesse

L'exception safety assume rilevanza quando almeno una delle condizioni suddette non é vera.

💡 non prendere due risorse in un blocco try catch

Ideomi RAII - RRID?

Resurce Acquisition Is Initialization: quando acquisisco una risorsa lo faccio in un costruttore

Resurce Release Is Destruction:

```
class Gestore_Risorsa{
    private:
        Risorsa *res_ptr;
    public:
        // Costruttore : acquisisce la risorsa (RAII)
        Gestore_Risorsa() : res_ptr(acquisisce_risorsa_exc()){}
```

```
        // Distruttore : rilascia la risorsa (RRID)
        Gestore_Risorsa(){
            restituisce_risorsa(res_ptr);
        }

        // Disabilitazione delle copie
        Gestore_Risorsa(const Gestore_Risorsa&) = delete;

}
```

Uso il gestore della risorsa per l'acquisizione delle risorsa e il suo rilascio.

```
void codice_utente(){
    Gestore_Risorsa r1;
    usa_risorsa_exc(r1.get());
    {
        Gestore_Risorsa r2;
        usa_risorsa_exc(r1.get(), r2.get());
    }// vado fuori scope e distruggo r2
    Gestore_Risorsa r3;
    usa_risorsa_exc(r1.get(), r3.get());
 }
```

💡 quando codifico un **distruttore** non posso **mai** far uscire un'eccezione fuori

Scrivere una funzione `no_exc()` può aiutare

I distruttori devono essere `no_exc()`

🚫 `Gestore_Risorsa *r1 = new Gestore_Risorsa();`
In questo caso dovrei cancellare il puntatore ma è quella situazione che vogliamo evitare

Exception Safety

https://www.stroustrup.com/except.pdf

Esistono tre diversi livelli di excetion safety:

- Base
- Forte (strong)
- Nothrow

Livello Base

Una porzione di codice (una funzione o una classe) si dice exception safe a livello base se, anche nel caso in cui si verifichino delle eccezioni durante la sua esecuzione

1. non si hanno perdite di risorse
2. si è neutrali rispetto alle eccezioni
3. anche in caso di uscita in modalità eccezionale, gli oggetti sui quali si stava lavorando sono distruggibili senza causare comportamenti non definiti.

Livello Forte (strong)

Ha le proprietà del livello base, ma nel terzo punto fa qualcosa di più, cioè l'atomicità delle operazioni

Atomicità: garanzia che gli oggetti manipolati non sia alterato

Livello Nothrow

Una funzione è northrow se la sua esecuzione è garantita non terminare in modalità eccezionale.

Questo livello lo si raggiunge in un numero limitato:

- Quando la funzione è così semplice che NON c'è la possibilità di generare eccezioni (esempio, assegnamento di tipi built-in);
- Quando la funzione è in grado di gestire completamente al suo interno eventuali eccezioni, risolvendo eventuali problemi e portando comunque a termine con successo l'operazione richiesta;
- Quando la funzione, di fronte a eventuali eccezioni interne, nell'impossibilità di attuare azioni correttive, determina la terminazione di tutto il programma; questo è il caso delle funzioni che sono dichiarate (implicitamente o esplicitamente) `noexcept', come i *distruttori*: in caso di eccezione non catturata, viene automaticamente invocata la terminazione del programma. Intuitivamente, devono garantire il livello nothow i distruttori e le funzioni che implementano il rilascio delle risorse (non è

ipotizzabile ottenere l'exception safety se l'operazione di rilascio delle risorse può non avere successo).

Per usare Try Catch sulle risorse

```
Risorsa* r3 = acquisici_risorsa_exc();
try{ // blocco try che protegge la risorsa r3
    usa_risorsa_exc(r1,r3);
    restituisci_risora(r3);
} // fine try che protegge r3
catch(...){
    restituisci_risorsa(r3);
    throw; // exception neutrality
}
```

1. si crea un blocco try/catch per ogni singola risorsa acquisita
2. il blocco si apre subito *dopo* l'acquisizione della risorsa (se l'acquisizione fallisce, non c'e' nulla da rilasciare)
3. la responsabilità del blocco try/catch e' di proteggere *quella* singola risorsa (ignorando le altre)
4. al termine del blocco try (prima del catch) va effettuata la "normale" restituzione della risorsa (caso NON eccezionale)
5. la clausola usa "…" per catturare qualunque eccezione: non ci interessa sapere che errore si e' verificato (non e' nostro compito), dobbiamo solo rilasciare la risorsa protetta
6. nella clausola catch, dobbiamo fare due operazioni:
    a. lasciare la risorsa protetta
    b. rilasciare l'eccezione catturata (senza modificarla) usando l'istruzione throw

    Il rilascio dell'eccezione catturata garantisce la "neutralità rispetto alle eccezioni": i blocchi catch catturano le eccezioni solo temporaneamente, lasciandole poi proseguire.

Smart pointers

L'uso dei semplici puntatori forniti dal linguaggio (detti anche puntatori "raw" o "naked" o addirittura "dumb", in contrapposizione a quelli "smart") si presta a una serie di possibili errori di programmazione nei quali può incappare anche un programmatore esperto.

La libreria standard offre classi templatiche che fornisce diverse tipologie di puntatori “smart”: **unique_ptr**, **shared_ptr** e **weak_ptr**. Le tre classi templatiche sono definite nell’header file <memory>.

> 💡 **Nota**
>
> I puntatori smart forniti dalla libreria standard sono concepiti per memorizzare puntatori a memoria allocata dinamicamente sotto il controllo del programmatore; non si possono utilizzare per la memoria ad allocazione statica o per la memoria ad allocazione automatica (sullo stack di sistema).

std::unique_ptr

Un `unique_ptr<T>` e’ un puntatore smart ad un oggetto di tipo T.

In particolare, `unique_ptr` implementa il concetto di puntatore “owning”, ovvero un puntatore che si considera l’unico proprietario della risorsa. Intuitivamente, lo smart pointer dovrà rilasciare la risorsa a lavoro finito, quindi gestirla in modo corretto.

Esempio:

```
#include<memory>

void foo(){
    std::unique_ptr<int> pi(new int(42));
    std::unique_ptr<double> pd(new double(3.1415));
    *pd *= *pd; // si dereferenzia come un puntatore
    // altri usi...
} // qui termina il tempo di vita di pi e pd e viene rilasiata l
```

> 📖 **Caratteristiche**
>
> Una caratteristica degli `unique_ptr` e’ il fatto di NON essere *copiabili*, ma di essere (solo) *spostabili*. La copia e’ impedita in quanto violerebbe il requisito di unicità del gestore della risorsa; lo spostamento e’ invece consentito, in quanto si trasferisce la proprietà’ della risorsa al nuovo gestore.

Esempio:

```
void foo(std::unique_ptr<int> pi);

void bar(){
    std::unique_ptr<int> pj(new int(42));
    // foo(pj);         // errore di compilazione: copia non amm
    foo(std::move(pj)); // ok: spostamento ammesso
    // dopo lo spostamento, pj non gestisce nessuna risorsa
}
```

La classe fornisce poi metodi per poter interagire con i puntatori “raw”, da usarsi nel caso in cui ci si debba interfacciare con codice che, per esempio, era stato sviluppato prima dell’adozione dello standard C++11.

Esempio:

```
std::unique_ptr<int> pi; // pi non gestisce (ancora) una risorsa
int* raw_pi=new int(42);
pi.reset(raw_pi);        // NON devo invocare la delete su raw_p
int* raw_pj=pi.get();    // NON devo invocare la delete su raw_p
int* raw_pk=pi.release();// devo invocare la delete su raw_pk
```

std::shared_ptr

Uno `shared_ptr<T>` e’ un puntatore smart ad un oggetto di tipo T.

La risorsa e’ condivisa da altri `shared_ptr` di tipo T.

A livello di implementazione, la copia causa l’incremento di un contatore del numero di riferimento alla risorsa (reference counter); quando uno shared_ptr viene distrutto, decrementa il reference counter associato alla risorsa e, se si accorge di essere rimasto l’unico shared_ptr ad avere ancora accesso, ne effettua il rilascio.

📖 Gli `shared_ptr` consentono la copia.
Viene copiato solo il puntatore della risorsa, quindi non avviene una deep copy

Esempio:

```
#include<memory>
void foo(){
    std::shared_ptr<int> pi;
```

```
    {
        std::shared_ptr<int> pj(new int(42)); // ref counter=1
        pi=pj;   // condivisione risorsa, ref counter=2
        ++(*pj); // uso risorsa condivisa: nuovo valore 43
        ++(*pj); // uso risorsa condivisa: nuovo valore 44
    } // distruzione pj, ref counter=1
    ++(*pi);   // // uso risorsa condivisa: nuovo valore 45
} // distruzione pi, ref counter=0, rilascio risorsa
```

💡 **Osservazione di efficienza (I template di funzione std::make_shared e std::make_unique)**

Un puntatore shared deve interagire con due componenti: la risorsa e il "blocco di controllo" della risorsa (una porzione di memoria nella quale viene salvato anche il reference counter).
Per motivi di efficienza, sarebbe bene che queste due componenti fossero allocate con una singola operazione: questa e' la garanzia offerta dalla `std::make_shared`

Esempio:

```
void bar(){
    auto pj=std::make_shared<int>(42);
    auto pj=std::make_shared<double>(3.1415);
}
```

Esempio:

```
void bar(std::shared_ptr<int> pi,
                std::shared_ptr<int> pj);

void foo(){
    // codice NON exception safe
    bar(std::shared_ptr<int>(new int(42)),
            std::shared_ptr<int>(new int(42)));
            // qui possiamo avere memory leak in quanto la chiam
            // un ordine non specificato

    // codice exception safe
```

```
    bar(std::make_shared<int>(42),
            std::make_shared<int>(42));
}
```

std::weak_ptr

Un problema che si potrebbe presentare quando si usano gli shared_ptr (piu' in generale, quando si usa qualunque meccanismo di condivisione di risorse basato sui reference counter) e' dato dalla possibilità di creare insiemi di risorse, che puntandosi reciprocamente tramite `shared_ptr` , formino una o piu' strutture cicliche.

In questo caso, le risorse comprese in un ciclo mantengono dei reference count positivi anche se non sono piu' raggiungibili a partire dagli `shared_ptr` ancora accessibili da parte del programma, causando dei memory leak piu' complicati.

L'uso dei `weak_ptr` e' pensato per risolvere questi problemi.

Un `weak_ptr` e' un puntatore a una risorsa condivisa che però non partecipa attivamente alla gestione della risorsa stessa: la risorsa viene quindi rilasciata quando si distrugge l'ultimo `shared_ptr` , anche se esistono dei `weak_ptr` che indirizzano.
Ciò significa che un
`weak_ptr` non può accedere direttamente alla risorsa: prima di farlo, deve controllare se la risorsa e' ancora disponibile. Il modo migliore per farlo e' mediante l'invocazione del metodo `lock()` , che produce uno shared_ptr a partire dal `weak_ptr` : se la risorsa non e' piu' disponibile, lo `shared_ptr` ottenuto conterrà il puntatore nullo.

Esempio:

```
void maybe_print(std::weak_ptr<int> wp){
if(auto sp2=wp.lock()) // lock() "chiede" se sta puntando la ris
    std::cout<<*sp2; // nel caso ci fosse la risorsa la spampa
else
    std::cout<<"non piu' disponibile"
}

void foo(){
std::weak_ptr<int> wp;
{
auto sp=std::make_shared<int>(42);
```

```
wp=sp; // wp non incrementa il reference count della risorsa
*sp=55;
maybe_print(wp); // stampa 55
} // sp viene distrutto, insieme alla risorsa

maybe_print(wp); // stampa "non piu' disponibile"
}
```

Categorie di espressioni : lvalue e rvalue

Lvalue deve stare a sinistra dell'assegnamento

Rvalue deve stare a destra dell'assegnamento

Le espressioni del C++ possono essere classificate in

1. **lvalue** (left value): identita' non spostabile
2. **glvalue** (generalized left value): identita'
3. **xvalue** (expiring lvalue): identita' spostabile
4. **rvalue** (right value): spostabile
5. **prvalue** (primitive rvalue): non identità spostabile

L'unione di lvalue e xvalue forma i glvalue (generalized lvalue).

L'unione di xvalue e prvalue forma gli rvalue (right value).

```
Matrix foo1(){
    Matrix m;
    ...
    return m; // espressione è un xvalue
}

Matrix foo2(){
    Matrix m1;
    m1 = foo1(); // xvalue è un temporaneo
}
```

💡 Lvalue copio
Rvalue sposto

```
// Costruzione per spostamento (C++11)
  Gestore_Risorsa(Gestore_Risorsa&& y)
    : res_ptr(y.res_ptr) {
    y.res_ptr = nullptr;
  }

// Assegnamento per spostamento (C++11)
  Gestore_Risorsa& operator=(Gestore_Risorsa&& y) {
    restituisci_risorsa(res_ptr);
    res_ptr = y.res_ptr;
    y.res_ptr = nullptr;
    return *this;
  }
```

Funzione che trasforma un valore lvalue in un rvalue

```
bar(std::move(m)); // m sarebbe lvalue ma con move(m) la trasfor
```

In realtà `move()` non “muove” nulla. Lo spostamento vero e proprio viene effettuato durante il passaggio del parametro.

I template di C++

Template

I template sono schemi, template di classi e funzioni

```
template <typename T> struct S // dichairazione pura di template

template <typename T> // dichiarazione e definizione
struct S {
  T t;
};
```

```
template <typename T> // dichairazione pura di template di funzi
T add(T t1, T t2);

template <typename T> // definizione di template di funzione
T add(T t1, T t2) {
  return t1 + t2;
}
```

💡 **Nota**: differenza tra classe e struct
Nella class se non si specifica l'accesso ne public ne private ne protected, i dati membri rimangono private, mentre nella struct rimangono public.

💡 **Nota:** differenza tra template e tipo di dato
Il template è un meccanismo per produrre tipi dato; quando nel typename ci metto un tipo concreti ad esempio int allora il template creerà il tipo S di int

Template di funzione

Un template di funzione e' un costrutto di linguaggio c++ che consente di scrivere un *modello* (schema) parametrico per una funzione.

Esempio:

```
// dichiarazione pura di un template di funzione
template <typename T>
T max(T a, T b):

/// definizione di un template di funzione
template <typename T>
T max(T a, T b){
    return (a > b)? a:b; // interpretazione: a e' maggiore di b?
}
```

Parametri del template

La lista dei parametri può contenere un numero arbitrario, separati da virgola; oltre ai parametri che sono nomi di tipo, esistono anche tipologie *valori, template*.

> 💡 **Nota**
> Per convenzione si usano nomi maiuscoli per i parametri di tipo; il nome “T” e’ comunque arbitrario (e può essere cambiato a piacere).

Esempio:

```
void foo(int a);
void foo(int b); // dichiara la stessa funzione

template <typename T>
T max(T a, T b);
template <typename U>
U max(U x, U y); // dichiara lo stesso template max
```

Instanziazione di un template di funzione

Dato un template di funzione, e’ possibile “generare” da esso una o piu’ funzioni mediante il meccanismo di instanziazione (del template): l’instanziazione fornisce un *argomento* (della tipologia corretta) ad ognuno dei parametri del template.

L’instanziazione avviene spesso in maniera *implicita* , quando si fa riferimento al nome del template allo scopo di “usarne” una particolare istanza.

Esempio:

```
void foo(int i1, int i2, double d1, double d2){
    // instanziazione della funzione
    // int max<int>(int, int);
    int m=max<int>(i1, i2);

    // instanziazione della funzione
    // double max<doubel>(doubel, double);
    double d=max<double>(d1, d2);
}
```

Quando si istanzia un template di funzione, e’ possibile evitare la sinstassi esplicita per gli argomenti del template, lasciando al compilatore il compito di *dedurre* tali argomenti a partire dal tipo degli argomenti passati alla chiamata di funzione

Esempio:

```
void foo(char c1, char c2){
    // instanzione della funzione
    // char max<char>(char, char);
    int m=max(c1, c2);
    // il legame T=char viene dedotto dal tipo degli argomenti c
    // si noti che il tipo di m (int) non influisce sul processo
}
```

Il processo di deduzione potrebbe fallire a causa di ambiguità:

Esempio:

```
void foo(double d, int i){
    int m=max(d, i); // errore
    // il compilatore non puo' dedurre un unico tipo T coerente
    int m=max<int>(d, i); // ok: evito la deduzione
}
```

💡 **Nota**
E’ opportuno sottolineare la differenza sostanziale tra un template di funzione e le sue possibili instanziazioni. In particolare:
Un template di funzione NON e’ una funzione (e’ un “generatore” di funzione); una istanza di template di funzione e’ una funzione

NON posso prendere l’indirizzo di un template di funzione (posso prendere l’indirizzo di una istanza specifica); non posso effettuare una chiamata di un template (chiamo una istanza specifica); non posso passare un template come argomento ad una funzione (passo una istanza specifica, che corrisponde a passarne l’indirizzo usando il type decay).

❗ Spesso si parla di “chiamata di un template di funzione” per indicare la chiamata di una sua specifica istanza

Specializzazione esplicita di un template di funzione

Capita a volte che la definizione di un template di funzione sia adeguata per molti, ma non per tutti i casi di interesse; ad esempio, il codice scritto potrebbe fornire un risultato ritenuto sbagliato quando ai parametri del template sono associati argomenti particolari.

Ad esempio, il template `max` può essere istanziato anche con il tipo “const char*”, ottenendo una funzione che restituisce il massimo dei due puntatori passati, quando invece, le intenzioni dell’utente era di fare un confronto lessicografico tra due stringhe stile C.

Esempio:

```
// definizione del template
template <typename T>
T max(T a, T b){
    return (a > b) ? a:b;
}

// specializzazione esplicita (per T=const char*) del template
template<>
const char* max<const char*>(const char* a, const char* b){
    return strcmp(a, b)>0;
}
```

Nota

Sarebbe stato possibile evitare la specializzazione del template e fornire la versione specifica per i const char* come funzione “normale”, sfruttando l’overloading di funzione:

`const char* max(const char* a, const char* b);`
Diventa importante capire come si comporta il meccanismo di risoluzione dell’overloading in questi casi

Instanziazione esplicite di template

Sono previste due sintassi, corrispondenti a due casi d’uso distinti (che tipicamente occorrono in unità di traduzione diverse facenti parte della stessa applicazione).

1. Dichiarazione di instaziazione esplicita

   ```
   extern template
   float max(float a, float b);
   ```

2. Definizione di instaziazione esplicita

```
template
float max(float a, float b);
```

Template di classe

Un template di classe e' un costrutto del linguaggio che consente di scrivere un "modello" parametrico per un classe.

Quasi tutti i concetti esposti per il caso di template di funzione possono essere applicati alle classi: nel seguito si sottolineano le differenze (poche ma importanti).

Esempio:

```
// dichiarazione pura di un template di classe
// (nota: il nome T del parametro potrebbe essere omesso)
template <typename T>
class Stack;

// definizione di un template di classe
template <typename T>
class Stack{
public:
    /*...*/
    void push(const T& t);
    void pop();
    /*...*/
};
```

Nel caso di template di classe e' ancora piu' importante distinguere tra il nome del template (Stack) e il nome di una specifica istanza (per esempio, Stack<std::string>). Infatti, per i template di classe NON si applica la deduzione dei parametri del template: la lista degli argomenti va indicata obbligatoriamente.

```
Stack<int> s1; // instanziazione implicita del tipo Stack<int>
                              // in particolare, del costruttore
Stack s2;      // errore: non viene dedotto il tipo T=int
auto s2=s1;    // ok: il C++11 ha introdotto la deduzione di tip
               // dall'inizializzatore, usando auto; viene anche
               // instanziato implicitamente il costruttore di c
```

L’unico caso in cui e’ lecito usare il nome del template di classe per indicare il nome della classe ottenuta mediante instanziazione e’ all’interno dello scope del template di classe stesso.

Esempio:

```
template <typename T>
class Stack{
    /*...*/
    /* qui gli usi di Stack sono abbreviazioni (lecite) di Stack
    Stack& operator=(const Stack&);
}; // usicamo dallo scope di classe

// definizione (out-of-line)
template <typename T>
Stack<T>& // il tipo di ritorno e' fuori scope di classe, devo s
Stack<T>::operator=(const Stack& y){ // parametro in scope di cl
    Stack tmp=y; // in scope di classe, e' sufficiente Stack
    swap(tmp);
    return *this;
}
```

Instanziazione one demand (al bisogno)

E’ importante sottolineare che, quando si istanzia implicitamente una classe templatica, vengono generate solo le funzionalità necessarie per il funzionamento del codice che causa l’instanziazione.

Esempio:

```
template <typename T>
class Stack{
public:
    /*...*/
    void push(const T& t);
    void pop();
    /*...*/
};

/*...*/
```

```
Stack<int> s1;
```

Questa scelta del linguaggio ha aspetti positivi e negativi:

Negativo: le funzionalità NON testate (e quindi non istanziate) potrebbero addirittura generare errori di compilazione al momento dell'instanziazione da parte dell'utente;

Positivo: posso usare un sottoinsieme delle funzionalità della classe in stanziandola con argomenti che soddisfano solo i requisiti di quelle funzionalità, ovvero ogni volta che uso un metodo quel metodo ha delle precondizioni

Instanziazione e specializzazione di template di classe

Come nel caso di template di funzione, anche i template di classe possono essere istanziati (esplicitamente o implicitamente) e specializzati.

Un esempio di specializzazione *totale* di template di classe e' fornito all'interno dell'header file standard `<limits>`, che fornisce il template di classe `std::numeric_limits` , attraverso il quale si possono per esempio ottenere informazioni sui tipi built-in:

Esempio d'uso:

```
#include <limits>
int foo(){
    long minimo=std::numeri_limits<long>::min()
    long massimo=std::numeri_limits<long>::max()
    bool char_con_segno=std::numric_limits<char>::is_signed;
}
```

Nell'header file `limits` troviamo, tra le altre cose, le specializzazioni totali che consentono di rispondere alle interrogazioni dell'utente:

```
[...]
    /// numeric_limits<char> specialization.
    template<>
        struct numeric_limits<char>
[...]
    /// numeric_limits<long> specialization.
    template<>
        struct numeric_limits<long>
[...]
```

Programmazione generica in C++

I template vengono usati in C++ per realizzare il “polimorfismo statico”:

- Si parla di “polimorfismo in quanto si scrive una sola versione del codice (template) che però viene utilizzata per generare tante varianti (istanze) e quindi può assumere tante forme concrete”
- Il polimorfismo e’ “statico” in quanto la scelta delle istanze da generare avviene staticamente, a tempo di compilazione; cioè non avviene a run-time, come nel caso del polimorfismo “dinamico”.

> 💡 Polimorfismo statico = programmazione sui template

I contenitori

Un contenitore e’ una classe che ha lo scopo di contenere una collezione di oggetti (spesso chiamati elementi del contenitore); essendo spesso richiesto che il tipo degli elementi contenuti sia arbitrario (scelto dal programmatore), i contenitori sono tipicamente realizzati mediante template di classe, che si differenziano a seconda dell’organizzazione della collezione di oggetti e delle operazioni fondamentali che si intendono supportare (in maniera efficiente) su tali collezioni.

I contenitori sequenziali

I contenitori sequenziali forniscono accesso ad una sequenza di elementi, organizzati in base alla loro posizione (il primo elemento, il secondo, il terzo, ecc…). L’ordinamento degli oggetti nella sequenza non e’ stabilito in base ad un criterio di ordinamento stabilito a priori, ma viene dato dalle specifiche operazioni di inserimento e rimozione degli elementi (effettuati a partire da posizioni determinate della sequenza).

I contenitori standard sono:

```
std::vector<T>
```

```
std::deque<T>
```

```
std::list<T>
```

```
std::forward_list<T>
```

Contenitori associativi

I contenitori associativi sono contenitori che organizzano gli elementi al loro interno in modo da facilitare la ricerca in base al valore di “chiave”.

```
std::set<Key, Cmp>
std::multiset<Key, Cmp>
std::map<Key, Mapped, Cmp>
std::multimap<Key, Mapped, Cmp>
std::unordered_set<Key, Hash, Equal>
std::unordered_multiset<Key, Hash, Equal>
std::unordered_map<Key, Mapped, Hash, Equal>
std::unordered_multimap<Key, Mapped, Hash, Equal>
```

Proprietà contenitori associativi:

1. La presenza (o assenza) negli elementi di ulteriori informazioni, oltre alla chiave usata per effettuare le associazioni;

   Se il tipo elemento e’ formato solo dalla chiave (Key), allora abbiamo i contenitori detti “insiemi” (set); altrimenti abbiamo i contenitori dette “mappe” (map), associano valori del tipo Key a valori del tipo Mapped; in particolare, nel caso degli insiemi, il tipo degli elementi contenuti e’ “const key”, mentre nel caso delle mappe il tipo degli elementi contenuti e’ la coppia “std::pair<const Key, Mapped>”.

2. La possibilita’ o meno di memorizzare nel contenitore piu’ elementi con lo stesso valore per la chiave.

   Nel caso sia possibile memorizzare piu’ elementi con lo stesso valore per la chiave, abbiamo le versioni “multi” dei contenitori (multinsiemi, multimappe, ecc).

3. Il fatto che l’organizzazione interna del contenitore sia ottenuta mediante un criterio di ordinamento delle chiavi (il tipo Cmp) oppure attraverso una opportuna funzione di hashing (il tipo Hash);

   Nel primo caso, abbiamo la possibilità di scorrere gli elementi nel contenitore in base al criterio di ordinamento; l’implementazione interna deve garantire che la ricerca di un valore con una determinata chiave possa essere effettuata eseguendo un numero di confronti O(log_n).

Nel secondo caso (funzione di hashing) si ottengono invece i contenitori ”unordered”: questi organizzano gli elementi in una tabella hash per cui quando si scorrono non si presentano secondo un criterio di ordinamento “naturale”. L’implementazione interna garantisce che la ricerca di un valore con una determinata chiave abbia nel caso medio un costo costante: per fare questo, la funzione di hashing calcola una posizione “presunta” nella tabella hash e poi, usando la funzione di confronto per uguaglianza (il tipo Equal) si controlla se vi sono stati clash.

Criteri di confronto per le mappe

Due chiavi sono equivalenti se non e’ vero che la prima chiave e’ minore della seconda e non e’ vero che la seconda e’ minore della prima

```
if( !(k1<k2) && !(k2<k1) ) k1=k2;
```

Nota
Gli algoritmi
non possono lavorare direttamente sui contenitori

Pseudo-contenitori/quasi contenitori

```
std::array<T, N>
```

```
std::string
```

```
std::bitset<N>
```

Nota

Predicati = funzioni che restituiscono un booleano

Adattatori

Oltre ai contenitori, nella libreria sono forniti gli “adattatori”; questi forniscono ad un contenitore esistente una interfaccia specifica per usarlo “come se” fosse un determinato tipo di dato.

> 💡 **Esempi di adattatori**
>
> `std::stack<T, C>` e `std::queue<T, C>`
> Che forniscono le classiche strutture dati di pila (LIFO) e coda (FIFO). Al loro interno, usano un altro contenitore standard (il tipo C; la scelta di default e’ `std::deque<T>`, sia per le pile che per le code)

Esiste anche l’adattatore `std::priority_queue<T, C, Cmp>` per le code con priorità (la classica struttura dati heap), nelle quali la priorità; tra gli elementi e’ stabilita dal criterio di confronto Cmp.

In questo caso, il contenitore C usato per default e’ uno `std::vector<T>`.

> 💡 **NOTA**
> gli adattatori **NON** implementano il concetto di sequenza; in particolare, **NON** forniscono i tipi iteratore e i corrispondenti metodi `begin()` e `end()`

Il concetto di iteratore

Molti algoritmo generici della libreria standard lavorano sul concetto di sequenza. Il concetto di iteratore, che prende spunto dal puntatore, fornisce un modo efficace per rappresentare varie tipologie di sequenze, indipendentemente dal tipo concreto usato per la loro implementazione.

Gli iteratori si classificano in 5 categorie che si differenziano per le operazioni supportate e per le corrispondenti garanzie fornite all’utente.

- Iteratori di input
- Iteratori forward
- Iteratori bidirezionali
- Iteratori random access
- Iteratori di output

Iteratori di input

Consentono di effettuare le seguenti operazioni:

```
++iter         // avanzamento di una posizione nella sequenza
iter++         // avanzamento postfiso (non usarlo: preferire la
*iter          // accesso (in sola lettura) all'elemento corrent
iter->m        // equivalente a (*iter).m dove si assume che l'e
iter1 == iter2 // confronto (per uguaglianza) tra iteratori: tip
iter1 != iter2 //confronto per disuguaglianza
```

Esempio:

```
#include <iterator>
#include <iostream>

int main() {

  // uso di iteratori per leggere numeri double da std::cin

  std::istream_iterator<double> i(std::cin); // inizio della (ps
  std::istream_iterator<double> iend;        // fine della (pseu

  // scorro la sequenza, stampando i double letti su std::cout
  for ( ; i != iend; ++i)
    std::cout << *i << std::endl;
}
```

Nota

Nel caso degli istream, l'iteratore che indica l'inizio della sequenza si costruisce passando l'input stream (std::cin), mentre quello che indica la fine della sequenza si ottiene col costruttore di default.

Quando si opera con un iteratore di input occorre tenere presente che l'operazione di incremento potrebbe invalidare eventuali altri iteratori definiti sulla sequenza

Esempio:

```
std::istream_iterator<double> i(std::cin); // inizio della (pseu
auto j = i;
// ora j e i puntano entrambi all'elemento corrente
std::cout << *i; // stampo l'elemento corrente
std::cout << *j; // stampo ancora l'elemento corrente

++i;             // avanzo con i: questa operazione rende j *inv
std::cout << *j; // errore: comportamento NON definito
```

Iteratori forward

Iteratori che vanno solo in avanti, fanno quello che fanno gli iteratori di input ma sono piu' potenti ,possono lavorare anche in scrittura sempre che l'oggetto non sia marcato **const.**

La differenza piu' importante e' che gli iteratori forward nelle operazioni di avanzamento non invalida eventuali altri iteratori che puntano a elementi precedenti nella sequenza.

Sono riavvolgibili

Esempi di iteratori forward sono quelli resi disponibili dal contenitore `std::forward_list`

Esempio:

```
#include <forward_list>
#include <iostream>

int main() {
  std::forward_list<int> lista = { 1, 2, 3, 4, 5 };

  // Modifica gli elementi della lista
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il ti
  // dell'iteratore usato, che sarebbe std::forward_list<int>::i
  for (auto i = lista.begin(); i != lista.end(); ++i)
    *i += 10;

  // Stampa i valori 11, 12, 13, 14, 15
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il ti
```

```
  // dell'iteratore usato, che sarebbe std::forward_list<int>::c
  for (auto i = lista.cbegin(); i != lista.cend(); ++i)
    std::cout << *i << std::endl;
}
```

Ad esempio l'algoritmo `is_sorted` ha bisogno di un iteratore forward

```
template <typename FwdIter>
bool
is_sorted(FwdIter first, FwdIter last){
    if(first==last)
        return true;
    FwdIter next=first;
    ++next;
    for(;next!=last;++first,++next){
        if(*next<*first)
            return false;
    }
    return true;
}
```

Anche l'algoritmo `replace`

```
template <typename FwdIter, typename T>
void
replace(FwdIter first, FwdIter last, const T& old_value, const T
    for(;first!=last;++first){
        if(*first==old_value)
            *first=new_value;
    }
}
```

Iteratori bidirezionali

Si comportano come gli iteratori forward ma consentono di tornare indietro

```
--iter
iter--
```

Esempio:

```
#include <list>
#include <iostream>

int main() {
  std::list<int> lista = { 1, 2, 3, 4, 5 };

  // Modifica gli elementi della lista
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il ti
  // dell'iteratore usato, che sarebbe std::list<int>::iterator
  for (auto i = lista.begin(); i != lista.end(); ++i)
    *i += 10;

  // Stampa i valori all'indietro
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il ti
  // dell'iteratore usato, che sarebbe std::list<int>::const_ite
  for (auto i = lista.cend(); i != lista.cbegin(); ) {
    --i; // Nota: è necessario decrementare prima di leggere
    std::cout << *i << std::endl;
  }

  // Potevo ottenere (più facilmente) lo stesso effetto usando
  // gli iteratori all'indietro
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il ti
  // dell'iteratore usato, che sarebbe std::list<int>::const_rev
  for (auto i = lista.crbegin(); i != lista.crend(); ++i)
    std::cout << *i << std::endl;
}
```

Iteratori random access

Iteratori piu’ potenti di tutti, che rimpiazzano con la loro potenza i puntatori.

Sanno fare tutto quello che sanno fare gli iteratori bidirezionali, ma in piu’ puo’ fare anche

```
iter += n       // sposta iter di n posizioni (in avanti se n è
                // all'indietro se n è negativo)
iter -= n       // analogo, ma sposta nella direzione opposta

iter + n        // calcola un iteratore spostato di n posizioni
```

```
                // (senza modificare iter)
n + iter        // equivalente a iter + n
iter - n        // analogo, ma nella direzione opposta

iter[n]         // equivalente a *(iter + n)

iter1 - iter2   // calcola la "distanza" tra i due iteratori, ov
                // il numero di elementi che dividono le due pos
                // Nota: i due iteratori devono essere definiti
                // stessa sequenza.

iter1 < iter2   // restituisce true se iter1 occorre prima di it
                // nella sequenza (che deve essere la stessa)
iter1 > iter2   // analoghi
iter1 <= iter2
iter1 >= iter2
```

Esempio:

```
#include <vector>
#include <iostream>

int main() {
  std::vector<int> vect = { 1, 2, 3, 4, 5, 6 };

  // Modifica solo gli elementi di indice pari
  for (auto i = vect.begin(); i != vect.end(); i += 2)
    *i += 10;

  // Stampa i valori 11, 2, 13, 4, 15, 6
  for (auto i = vect.cbegin(); i != vect.cend(); ++i)
    std::cout << *i << std::endl;
}
```

Iteratori di output

Permettono di fare solamente scritture sugli elementi di una sequenza

Le uniche operazioni consentite sono le seguenti:

```
++iter    avanzamento di una posizione nella sequenza
iter++    avanzamento postfisso (NON usarlo: preferire la forma

*iter     accesso (in sola scrittura) all'elemento corrente
```

💡 **Nota**

Viene data la possibilità di confrontare iteratori di output tra di loro, in quanto NON è necessario farlo: un iteratore di output assume che vi sia sempre spazio nella sequenza per potere fare le sue scritture; è compito di chi lo usa fornire questa garanzia e, se la proprietà è violata, si otterrà un undefined behavior

Esempio:

```
#include <iterator>
#include <iostream>

int main() {

  std::ostream_iterator<double> out(std::cout, "\n"); // posizio
  // Nota: non esiste una "posizione finale"
  // Nota: il secondo argomento del costruttore serve da separat
  // se non viene fornito si assume la stringa vuota ""

  double pi = 3.1415;
  for (int i = 0; i != 10; ++i) {
    *out = (pi * i);  // scrittura di un double usando out
    ++out;            // NB: spostarsi in avanti dopo *ogni* scr
  }

}
```

💡 L’iteratore di output per default e’ in modalità overwrite

Per cambiare la modalità da overwrite a inserimento, utilizziamo gli `inserter`

💡 Gli inseritori sono nell’header file `stl_iterator.h`

Il concetto callable

Un callable e’ qualunque cosa che può essere invocato come una funzione

Ad esempio, per l'algoritmo std::adjacent_find, che intuitivamente ricerca all'interno di una sequenza la prima occorrenza di due elementi adiacenti ed equivalenti, abbiamo le seguenti dichiarazioni (in overloading):

```
template <typename FwdIter>
FwdIter adjacent_find(FwdIter first, FwdIter last);

template <typename FwdIter, typename BinPred>
FwdIter adjacent_find(FwdIter first, FwdIter last, BinPred pred)
```

Nella prima versione, il predicato binario utilizzato per il controllo di equivalenza degli elementi è operator==

💡 **Nota**
Istanze diverse del template possono usare definizioni diverse (in overloading) di operator==, ma il nome della funzione usata per il controllo di equivalenza è fissato ("hard-wired").

La seconda versione consente invece di utilizzare un qualunque tipo di dato fornito dall'utente, a condizione che questo si comporti come predicato binario definito sugli elementi della sequenza.
Tenendo a mente il "duck typing", ci dovremmo quindi chiedere quali sono i modi legittimi di istanziare il parametro template BinPred; in altre parole, ci chiediamo quali siano i tipi di dato concreti ammessi per il parametro funzione "pred", cioè quelli che consentono di compilare correttamente il test.

📖 **Duck Typing**
Codice che
If it walks like a duck and it quacks like a duck, then it must be a duck

Quali sono i tipi di dati che possono in stanziare un callable?

- Puntatori a funzione
- Oggetti funzione
- Expression lambda (dal C++11)

Puntatori a funzione

Per esempio, quando usiamo il nome della funzione:

```
bool pari(int i);
```

per istanziare il predicato unario della std::find_if, il parametro typename UnaryPred viene legato al tipo concreto "bool (*)(int)" (puntatore ad una funzione che prende un argomento intero per valore e restituisce un bool).

> 💡 **Nota**
> Da un punto di vista tecnico, sarebbe pure possibile passare le funzioni per riferimento (invece che per valore), evitando il type decay ed ottenendo quindi un riferimento invece che un puntatore. Siccome questa alternativa NON porta alcun beneficio concreto (anzi,
> complica solo la comprensione del codice), è considerato pessimo stile

Oggetti funzione

Oltre alle vere e proprie funzioni, vi sono altri tipi di dato i cui valori possono essere invocati come le funzioni e che quindi, in base al "duck typing", soddisfano i requisiti del concetto callable.

In particolare, una classe che fornisca una definizione (o anche più definizioni, in overloading) del metodo "operator()" consente ai suoi oggetti di essere utilizzati al posto delle vere funzioni nella
sintassi della chiamata di funzione.

Esempio:

```
struct Pari {
  bool operator()(int i) const { // questa verisone fornisce fle
    return i % 2 == 0;
  }
};
```

```
int foo() {
  Pari pari;
  if (pari(12345)) ...
  ...
}
```

Nota

La dichiarazione di operator(), detto operatore parentesi tonde o anche operatore di chiamata di funzione, presenta due coppie di parentesi tonde: la prima fa parte del *nome* dell'operatore, la seconda fornisce la lista dei parametri per l'operatore

Spesso l'operatore è marcato const perché gli oggetti funzione sono spesso "stateless" (non hanno stato, cioè non hanno dei dati membro) e quindi non sono modificati dalle invocazioni dei loro operator()

Espressioni lambda

L’espressione lambda denota una funzione senza nome.

L’espressione lambda e’ data dalla sintassi:

```
[](const long& i) { return i % 2 == 0; }
```

dove si distinguono i seguenti elementi:

```
[]                      --> capture list (lista delle catture)
(const long& i)         --> lista dei parametri (opzionale)
{ return i % 2 == 0; }  --> corpo della funzione
```

📖 **Che significato ha una lmbda expression?**

Quando il compilatore incontra una lambda expression il compilatore fa:

- definisce una classe “anonima”, cioe’ una classe dotata di un nome univoco scelto dal sistema
- definisce all’interno della classe un metodo operator() che ha i parametri, il corpo e il tipo di ritorno specificati (o dedotti) dalla lambda expression
- creazione di un oggetto funzione “anonimo”

In pratica e’ come se il programmatore avesse scritto:

```
struct Nome_Univoco {
  bool operator()(const long& i) const { return i % 2 == 0; }
};

auto iter = std::find_if(v.begin(), v.end(), Nome_Univoco());
```

La lista delle catture e’ usata quando l’espressione lambda deve accedere a variabili locali visibili nel punto in cui viene creata (che e’ diverso dal punto in cui verrà invocata).

```
void foo(const std::vector<long>& v, long soglia) {
    auto iter = std::find_if(v.begin(), v.end(), [soglia](const
// ... usa iter
}
```

In questo caso la definizione della lambda e’ equivalente ad una classe nella quale le variabili catturate sono memorizzate in dati membro, inizializzati in fase di costruzione dell’oggetto funzione:

```
struct Nome_Univoco {
    long soglia;
  Nome_Univoco(long s) : soglia(s) { }
  bool operator()(const long& i) const { return i > soglia; }
};

auto iter = std::find_if(v.begin(), v.end(), Nome_Univoco(soglia
```

Si possono catturare più variabili, separate da virgole.

La notazione [soglia] è equivalente alla notazione [=soglia] e indica una cattura per valore.

Se invece si utilizza la notazione [&soglia] si effettua una cattura per riferimento (utile quando si vogliono evitare copie costose).
Nella lista delle catture è possibile indicare "this", catturando così (per valore) il puntatore implicito all'oggetto corrente; la sintassi è ammessa se la lambda è definita all'interno di un metodo (non-statico) di una classe, dove è effettivamente disponibile il puntatore this.

Esistono notazioni abbreviate per le "catture implicite":

```
[=]   --> cattura (implicitamente) ogni variabile locale usata n
          per valore
[&]   --> cattura (implicitamente) ogni variabile locale usata n
          per riferimento
[=, &pippo, &pluto]  --> cattura (implicitamente) per valore, tr
                         pippo e pluto che sono catturate per ri
[&, =pippo, =pluto]  --> cattura (implicitamente) per riferiment
                         pippo e pluto che sono catturate per va
```

💡 **Nota**
Il consiglio è di effettuare sempre catture esplicite, per maggiore leggibilità del codice

💡 **Nota**
L'espressione lambda crea un oggetto funzione anonimo
di tipo anonimo. E' comunque possibile dare un nome all'oggetto lambda, anche se non se ne conosce il tipo, sfruttando "auto" per effettuare la deduzione del tipo. Nell'esempio seguente, diamo un nome alla lambda per poterla usare più volte

```
void copia_corte(const std::vector<std::string>& v,
                 const std::list<std::string>& l,
                 unsigned max_size) {

  auto corta = [max_size](const std::string& s) {
                 return s.size() <= max_size;
               };

  std::ostream_iterator<std::string> out(std::cout, "\n");
  out = std::copy_if(v.begin(), v.end(), out, corta);
  out = std::copy_if(l.begin(), l.end(), out, corta);
}
```

Overloading e template di funzione

Esistono regole speciali nel caso in cui alcune funzioni siano state ottenute come in stanziamento o specializzazione di template di funzione.

❗ **I template di funzione NON sono funzioni.**

I template di funzione sono degli schemi per generare, mediante instanziazione, un numero potenzialmente illimitato di funzioni vere e proprie; ad esempio, possono prendere l’indirizzo di una istanza del template, ma non esiste un indirizzo di template.
Quindi quando si dice che un template di funzione e’ nell’insieme delle candidate nella risoluzione dell’overloading, in realtà si intende che una sua specifica istanza e’ candidata.

Ordinamento parziale dei template di funzione

I template di funzione con lo stesso nome e visibilità nello stesso scope, sono ordinati in modo parziale da una relazione di specificità.

```
template<typename T>
void foo(T t1, T t2){std::cout<<"template di funzione #3\n";}

template<typename T, typename U>
void foo(T t, U u){std::cout<<"template di funzione #4\n";}
```

📖 **Regola speciale per i template di funzione**

Nella risoluzione dell'overloading, le istanze dei template piu' specifici sono preferite a quelle dei template meno specifici.

Per decidere se una istanza di un template di funzione entra nell'insieme delle candidate, occorre verificare se e' possibile effettuare la deduzione dei parametri del template. Durante il processo di deduzione si possono applicare solo le corrispondenze esatte.

```
void foo(const char* s){std::cout<<"funzione #1\n";}

template<typename T>
void foo(T t){std::cout<<"template di funzione #2\n";}

template<typename T>
void foo(T t1, T t2){std::cout<<"template di funzione #3\n";}

template<typename T, typename U>
void foo(T t, U u){std::cout<<"template di funzione #4\n";}

template<typename T>
void foo(T* t1, T t2){std::cout<<"template di funzione #5\n";}

template<typename T, typename U>
void foo(T* t, U u){std::cout<<"template di funzione #6\n";}
------------------------------------------------------------
template<typename T>
void bar(T t, double d){std::cout<<"template di funzione #7\n";}
```

```
template<typename T>
void bar(T t, T t2){std::cout<<"template di funzione #8\n";}

/*
    Ordinamento parziale dei template:
        -#3 e' piu' specifico di #4
        -#5 e' piu' specifico di #4
        -#6 e' piu' specifico di #4
        -#5 e' piu' specifico di #6
*/



int main(){
    foo('a');             // #2: foo<char>(char)
    foo("aaa");           // #1: foo(const char*)

    int i=0;
    foo(i);               // #2: foo<int>(int)
    foo(i,i);             // #3: foo<int>(int,int)
    foo(i,&i);            // #4: foo<int,int>(int,int*)
    foo(&i,i);            // #5: foo<int,int>(int*,int)

    double d=0.0;
    foo(i,d);             // #4: foo<int,double>(int,double)
    foo(&d,i);            // #6: foo<double,int>(double*,int)

    short s=0;
    bar(i,d);             // #7: bar<int,double>
    bar(i,i);             // 8: bar<int,int>
    bar(i,s);             // #7: bar<int,short> perché fa la con
    bar(d,d);             // ambigua

}
```

💡 **Nota**
Se il processo di deduzione ha successo, l'istanza del template oltre ad essere candidata e' anche utilizzabile

📖 **Seconda Regola speciale per i template di funzione**

Quando si cerca la migliore funzione utilizzabile, si devono eliminare le istanza ottenute da templare di funzione meno specifici (che altrimenti causerebbero ambiguità).
Se anche dopo aver fatto questa rimozione continuiamo ad avere ambiguità, si eliminano dalle utilizzabili tutte le istante di template.
In questo modo, viene data la preferenza alle funzioni non templatiche.

Polimorfismo dinamico

📎 Argomenti da studiare e/o ripassare (i numeri dei capitoli si riferiscono allo Stroustrup):

- capitolo 20: Classi derivate (in particolare 20.3 e 20.4)
- capitolo 21: Gerarchie di classi
- capitolo 22: Informazioni sul tipo in esecuzione

Nel contesto di Polimorfismo statico il comportamento del codice viene deciso staticamente cioè a tempo di compilazione, nel polimorfismo dinamico abbiamo un'altra tecnica di programmazione che prevede un comportamento del codice che scriviamo a tempo di esecuzione.

💡 **Nota**
La risoluzione dell'overloading viene fatta dal compilatore a tempo di compilazione (ha informazioni solo di tipo statico), la risoluzione dell'overriding viene fatta a tempo di esecuzione

Classi derivate e relazione “IS-A”

Si consideri una classe Base e una classe Derived derivata pubblicamente dalla classe Base:

```
class Base {
  /* ... omissis ... */
};

class Derived : public Base {
  /* ... omissis ... */
};
```

La derivazione pubblica consente di effettuare in maniera implicita, le conversioni di tipo dette upcast, ovvero la conversione da un puntatore o riferimento per un oggetto Derived verso un puntatore o riferimento per un oggetto Base.

```
Base* base_ptr = new Derived;
```

> 💡 **Nota**
> Se la derivazione non fosse pubblica, cioè private o protected, tale conversione sarebbe legittima solo se effettuata nel contesto della classe derivata o all’interno di una funzione friend della classe.

Metodi virtuali e classi dinamiche

```
class Printer {
public:
  void print(const Doc& doc);
};

class FilePrinter : public Printer {
public:
  void print(const Doc& doc);
};

class NetworkPrinter : public Printer {
public:
```

```
  void print(const Doc& doc);
};
```

Supponiamo che il codice utente debba stampare alcuni documenti utilizzando una stampante e, non essendo interessato ai dettagli implementativi, utilizzi l'astrazione Printer nel modo seguente:

```
void stampa_tutti(const std::vector<Doc>& docs, Printer* printer
  for (const auto& doc : docs)
    printer->print(doc);
}
```

Il meccanismo dell’overriding viene attivato solo quando i metodi della classe base sono stati dichiarati essere “metodi virtuali” (cioè ridefinibili nelle classi derivate)

```
class Printer {
public:
  virtual void print(const Doc& doc);
};
```

> 💡 **Nota**
>
> Una classe e’ detta sovra classe dinamica (cioè sopporta il polimorfismo dinamico) se ha almeno un metodo marcato virtuale.

Avendo dichiarato virtuale il metodo:

```
void Printer::print(const Doc&);
```

una classe derivata da Printer che lo ridefinisce ne fa l’overriding.

> 💡 **Nota**
>
> Se il metodo non fosse stato dichiarato virtuale nella classe base, NON si avrebbe l’overriding ma si avrebbe l’hiding.

Se si usa C++11 o superiori e' consigliato usare la parola chiave “override”, da usarsi alla fine della dichiarazione.

```
class NetworkPrinter : public Printer{
public:
    void print(const Doc& doc) override;
}
```

L’uso di `override` e' utile perché causa un errore nel caso in cui nella classe base Printer non esistesse un metodo virtuale corrispondente.

Metodi virtuali puri e classi astratte

Per implementare un’interfaccia astratta si usa la sintassi “=0”.

```
class Printer {
public:
  virtual std::string name() const = 0;
  virtual void print(const Doc& doc) = 0;
};
```

Una classe che contenga metodo virtuali puri è detta "classe astratta" (in senso tecnico; spesso si usa dire che una classe è astratta anche in senso "metodologico", non tecnico, creando un po' di confusione). Il fatto che un metodo virtuale sia puro significa che ogni classe concreta che eredita dalla classe astratta Printer è tenuta a fare l'overriding del metodo; se NON fa l'overriding, il metodo rimane puro e quindi la classe derivata è anche essa una classe astratta (in senso tecnico).

> 💡 **Nota**
> NON è possibile definire un oggetto che abbia come tipo una classe astratta: questi possono solo essere usate come classi base per derivate altre classi (astratte o concrete).

Esempio:

```
Printer p;           // errore: Printer è astratta
NetworkPrinter np;   // ok, se NetworkPrinter ha effettuato l'ov
                     // di tutti i metodi virtuali puri di Print
```

I distruttori delle classi astratte

I distruttori delle classi astratte dovrebbero essere sempre dichiarati virtual e non dovrebbero mai essere metodi puri (ovvero, occorre fornirne l'implementazione). Ovvero, l'interfaccia di una classe dinamica astratta dovrebbe tipicamente avere la struttura seguente:

```
class Astratta {
public:
  // metodi virtuali puri
  virtual tipo_ritorno1 metodo1(parametri1) = 0;
  virtual tipo_ritorno2 metodo2(parametri2) = 0;
  virtual tipo_ritorno3 metodo3(parametri3) = 0;
  // ...

  // distruttore virtuale NON puro (definito, non fa nulla)
  virtual ~Astratta() {}
};
```

La ragione per questo modo di definire il distruttore è legata alla necessità di consentire una corretta distruzione degli oggetti delle classi concrete derivate dalla classe astratta. Infatti:

1. il distruttore della classe concreta invoca (implicitamente) il distruttore delle sue classi base, che quindi deve essere definito (cioè non può essere un metodo puro);
2. se il distruttore della classe astratta NON fosse virtuale, si avrebbero dei memory leaks.

Esempio:

```
class Astratta {
public:
  virtual void print() const = 0;
  ~Astratta() {} // distruttore errato: non è virtuale.
};

class Concreta : public Astratta {
  std::vector<std::string> vs;
```

```
public:
  Concreta() : vs(20, "stringa") {}
  void print() const override {
    for (const auto& s : vs)
      std::cout << s << std::endl;
  }
  // Nota: il distruttore di default sarebbe OK; lo ridefiniamo
  // per fargli stampare qualcosa, così che sia evidente il fatt
  // non è stato invocato.
  ~Concreta() { std::cout << "Distruttore Concreta" << std::endl
};

int main() {
  Astratta* a = new Concreta;
  a->print();
  // memory leak: non viene distrutto il vector nella classe con
  delete a; // invoca il distruttore di Astratta (che non è virt
}
```

Risoluzione overriding

Viene effettuata a tempo di esecuzione dal RTS (supporto a tempo di esecuzione). Si noti che, in ogni caso, a tempo di compilazione viene fatta la risoluzione dell'overloading nel solito modo. Affinché si attivi l'overriding occorre che:

1. il metodo invocato sia un metodo virtuale (esplicitamente o implicitamente, se ereditato da una classe base);
2. il metodo viene invocato tramite puntatore o riferimento (altrimenti non vi può essere distinzione tra il tipo statico e il tipo dinamico dell'oggetto e quindi si invoca il metodo della classe base);
3. almeno una delle classi lungo la catena di derivazione che porta dal tipo statico al tipo dinamico ha effettuato l'overriding (in assenza di overriding, si invoca il metodo della classe base);
4. il metodo NON deve essere invocato mediante qualificazione esplicita (la qualificazione esplicita causa l'invocazione del metodo come definito nella classe usata per la qualificazione).

Conversioni esplicite di tipo C++

Ci sono sei tipi diversi di conversioni esplicite:

1. static_cast
2. dynamic_cast
3. const_cast
4. reinterpret_cast
5. cast “funzionale”
6. cast stile C

Prima di considerare nel dettaglio le varie sintassi dei cast, vale la pena ragionare sui motivi (validi) per il loro uso.

Classificazioni delle motivazioni per l’uso di cast espliciti

Essendo conversioni esplicite di tipo, i cast dovrebbero essere utilizzati solo quando necessario. E’ possibile classificare gli usi dei cast in base alla motivazione, che frequentemente ricade in una di queste categorie:

a. Il cast implementa una conversione di tipo che NON e’ consentita dalle regole del linguaggio come conversione implicita, in quanto considerata una fonte di errori di programmazione; il programmatore, richiedendo la conversione esplicita, si assume la responsabilità della sua correttezza.

```
struct B { /* ... */ };
struct D : public B { /* ... */ };
D d;
B* b_ptr = &d;
// b_ptr è (staticamente, cioè a tempo di compilazione) un pu
// ma (dinamicamente, cioè a tempo di esecuzione) sta puntand
// oggetto di tipo D.
/* ... altro codice ... */
// Il programmatore forza il down-cast, prendendosi la respon
// di eventuali errori: se qualcuno nel frattempo avesse modi
// e questo non puntasse più ad un oggetto di tipo D, si otti
// Undefined Behavior.
D* d_ptr = static_cast<D*>(b_ptr);
```

b. Come nel caso precedente, ma il programmatore usa (in modo appropriato) un dynamic_cast allo scopo di controllare, a tempo di esecuzione, se la conversione richiesta e’ effettivamente consentita.

```
struct B { /* ... */ };
struct D : public B { /* ... */ };
void foo(B* b_ptr) {
    if (D* d_ptr = dynamic_cast<D*>(b_ptr)) {
  // posso usare d_ptr, che punta ad un oggetto di tipo D
  } else {
  // qui so che b_ptr NON sta puntando ad un oggetto di tipo
  }
}
```

c. Il cast NON e' strettamente necessario, ma il programmatore preferisce comunque la forma esplicita a scopo di documentazione, per tenere traccia esplicita della conversione di tipo effettuata e migliorare la leggibilità del codice; in altre parole, il programmatore ritiene che il cast sia necessario dal punto di vista metodologico (non e' necessario dal punto di vista tecnico).

```
double d = /* .... */;
// La conversione implicita double->int è ammessa, ma usando
// il programmatore vuole probabilmente attirare l'attenzione
// fatto che passando da un tipo floating point ad un tipo in
// tipicamente si perde informazione.
int approx = static_cast<int>(d);
```

d. Un caso speciale di un cast esplicito riguarda la conversione di una espressione al tipo void, che intuitivamente corrisponde a d una richiesta di “scartare” o “ignorare” il valore dell’espressione. Per convenzione, il cast a void si può usare per silenziare alcune segnalazioni di warning fornite dal compilatore.

```
// Il parametro size lo si usa solo nella assert e quindi, qu
// asserzioni NON sono attivate, il compilatore mi segnalereb
// il suo mancato uso mediante un warning;
// il cast esplicito serve a silenziare questo warning.
void foo(int pos, int size) {
  assert(0 <= pos && pos < size);
  static_cast<void>(size);
  /* ... codice che non usa size ... */
}
```

> 💡 **Nota**
> in questo caso spesso si usa un cast stile C:
>
> `(void) size;`
> In realtà, questo e' l'unico caso in cui l'uso di un cast stile C (in C++) e' tollerato: ogni altro uso e' considerato cattivo stile.

static_cast

Probabilmente, e' il cast utilizzato piu' frequentemente:

`static_cast<T>(expr)`

calcola un nuovo valore ottenuto dalla conversione del valore dell' espressione expr al tipo T. Il cast e' legittimo in uno dei casi seguenti (elenco parziale):

- e' legittima la corrispondente conversione implicita (caso banale);

```
double d = 3.14;
int approx = static<int>(d);
```

- e' legittima la costruzione diretta di un oggetto di tipo T passando expr come argomento;

```
Razionale r = static_cast<Razionale>(5);
```

- si effettua la conversione inversa rispetto ad una sequenza di conversione implicita ammissibile (con alcune restrizioni, per esempio non si possono invertire le trasformazioni di lvalue);

```
int i = 42;
void* v_ptr = &i;
int* i_ptr = static<int*>(v_ptr);
```

- il cast implementa un downcast in una gerarchia di classi;
- il cast implementa un cast da un tipo numerico ad un tipo enumerazione;
- il tipo destinazione e‘ void.

dynamic_cast

Il dynamic_cast e' uno degli operatori che forniscono il supporto per la cosiddetta RTTI (Run-Time Type Identification), cioè identificazione del tipo a tempo di esecuzione. I dynamic cast possono essere usati per effettuare conversioni all'interno di una gerarchia di classi legate da ereditarietà (singola o multipla). In particolare si possono effettuare:

- up-cast: conversione da classe derivata a classe base; effettuata raramente mediante `dynamic_cast` , in quanto e' una conversione consentita anche implicitamente e quindi non necessita della RTTI.
- down-cast: conversione da classe base a classe derivata; e' il caso piu' frequente di utilizzo del dyanmic cast, in quanto si sfrutta la RTTI per verificare che la conversione sia legittima.
- mixed-cast: caso particolare che si verifica quando si utilizza l'eredità multipla; consiste in uno spostamento nella gerarchia di ereditarietà ottenibile combinando up-cast e down-cast (da cui il nome di cast "misto"); siccome prevede comunque la presenza di down-cast, anche in questo caso si ha un uso banale della RTTI.

> 💡 Il dynamic cast si può applicare ai tipi puntatore (caso tipico) e anche ai tipi riferimento (caso raro), con una importante differenza semantica.

Supponiamo di avere la seguente gerarchia:

```
struct B { /* ... */ };
struct D1 : public B { /* ... */ };
struct D2 : public B { /* ... */ };
```

e di avere le funzione

```
void foo(B* b_ptr) { /* ... */ }
```

Se la classe B e' dinamica (contiene almeno un metodo virtuale) allora e' dotata delle informazioni per la RTTI e possono applicare cast dinamici ai puntatori per sapere se sono di un determinato tipo.

Esempio:

```
D1* d1_ptr = dynamic_cast<D1*>(b_ptr)
```

Dopo il cast dinamico va fatto un TEST:

```
if (d1_ptr != nullptr) {
  /* d1_ptr è valido */
}
if (d1_ptr) {
    /* equivalente: ho sfruttato la conversione a bool */
}
if (D1* d1_ptr = dynamic_cast<D1*>(b_ptr)) {
  /* equivalente: ho compattato cast e test */
}
```

const_cast

Il `const_cast` viene usato per rimuovere la qualificazione const. Tipicamente, si applica ad un riferimento o puntatore ad un oggetto qualificato const per ottenere un riferimento o puntatore ad un oggetto non qualificato (quindi modificabile).

```
void promessa_da_marinaio(const int& ci) {
  int& i = const_cast<int&>(ci);
  ++i;
}
```

Usando il `const_cast` potremo “rompere” il contratto stipulato con l’utente. Tra i pochi casi in cui può essere legittimo usare questo tipo di cast possiamo elencare i metodi di una classe che devono modificare la rappresentazione interna di un oggetto, senza però alterarne davvero il significato. Si tratta quindi di metodi che mantengono la “constness” a livello logico, pur violandola a livello fisico.

> 💡 **Nota**
> Alcuni usi di `const_cast` si potrebbero eliminare mediante l’utilizzo del modificatore `mutable` su alcuni metodi di una classe.

reinterpret_cast

Un `reinterpret_cast` può essere usato per effettuare le seguenti conversioni:

- da un tipo puntatore ad un tipo intero (sufficientemente grande da poter rappresentare il valore del puntatore);
- da un tipo intero/enumerazione ad un tipo puntatore;

- da un tipo puntatore (oppure riferimento) ad un altro tipo puntatore (oppure riferimento).

> 💡 **Nota**
> Non e' possibile usare un `reinterpret_cast` per rimuovere la qualificazione const (occorre usare il `const_cast`)

> 💡 **Nota**
> Nel caso di `reinterpret_cast` (diversamente dallo `static_cast`) le conversioni tra puntatori sono consentite anche quando i due tipi puntati NON sono in alcuna relazione tra di loro (in particolare, anche quando non fanno parte di una gerarchia di classi derivate). Quindi i `reinterpret_cast` sono una tra le forme di conversione piu' pericolose, in quanto i controlli di correttezza sono lasciati quasi completamente nelle mani del programmatore.

cast funzionale

La sintassi `T(expr)` oppure `T()`, dove T e' il nome di un tipo, viene spesso indicata come "cast funzionale"; intuitivamente, corrisponde alla costruzione diretta di un oggetto di tipo T, usando costruttore (nel secondo caso, il costruttore di default). Si parla di cast funzionale in quanto la sintassi si può applicare anche al caso dei tipi `builtin` (che in senso tecnico non sono dotati di costruttori). Nel caso di un tipo `builtin` , la forma T() produce la codetta zero-initialization.

```
template <typename T, typename U>
void foo(T t, U u) {
  if (t == T(u)) // cast funzionale
    // ...
}
```

cast stile C

Sintassi `(T) expr`

Il loro uso e' considerato cattivo stile (tranne il caso di cast a void per sopprimere warning del compilatore), perché:

- sono difficili da individuare nel codice mediante ricerca testuale;
- non differenziano le diverse tipologie di cast.

> 💡 **Nota**
> Con i cast stile C si possono simulare static_cast, const_cast e reinterpret_cast, ma NON i dynamic_cast (in particolare, non hanno accesso a informazioni RTTI e quindi non effettuano nessun controllo a run-time).

Principi progettazione object oriented

Con l'acrostico SOLID si indentificano i "primi 5 principi" della progettazione object oriented. Lo scopo dei principi e' di fornire una guida verso lo sviluppo di progetti che siano piu' "flessibili", cioè progetti per i quali sia relativamente semplificato effettuare modifiche in termini di:

- manutenzione delle funzionalità esistenti;
- estensione delle funzionalità supportate.

SOLID

S ⇒ SRP (Single Responsibility Principle)

O ⇒ OCP (Open-Closed Principle)

L ⇒ LSP (Liskov *(Barbara Liskov)* Substitution Principle)

I ⇒ ISP (Interface Segragation Principle)

D ⇒ DIP (Dependency Inversion Principle)

> 💡 Se il codice rispetta i principi:
>
> - si comportera' meglio in certi contesti
> - facile da mantenere/estendere

SRP (Single Responsibility Principle)

Si tratta di un principio di validità generale (cioè, non e' limitato al caso della progettazione object oriented) che dice, intuitivamente, che ogni porzione di software che progettiamo e implementiamo (una classe una funzione ecc…) dovrebbe avere in

carico una sola responsabilità. Si dice che una classe dovrebbe avere un solo “motivo per essere modificata”: se esistono piu’ motivi distinti, questo e’ indice che la classe si assume piu’ responsabilità e quindi dovrebbe essere suddivisa in piu’ componenti, ognuno dei quali caratterizzato da una singola responsabilità.

Il rispetto del principio porta a un codice piu’ mantenibile e, in linea di massima, pi’ facile da riutilizzare.

Esempio:

> 📌 una classe che deve manipolare una pluralità di risorse (in maniera exception safe) non dovrebbe prendersi carico direttamente della corretta gestione dell’acquisizione e rilascio delle singole risorse. Piuttosto, dovrebbe *delegare* questo compito ad opportune classi gestore (una classe gestore per ogni tipologia distinta di risorsa) e focalizzarsi sull’uso appropriato delle risorse.

OCP (Open-Closed Principle)

Il principio “aperto-chiuso” e’ forse il piu’ conosciuto dei principi SOLID. Ne sono state proposte due varianti:

- la prima si fa risalire al lavoro di Bertand Meyer (1988);
- la seconda (che e’ quella adottata nei principi SOLID) fu proposta da Robert C. Martin nel 1996, ma e’ di fatto una riformulazione di principi di progettazione proposti molti anni prima sotto altri nomi.

Usando le parole di Martin:

SOFWTARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.

> 📌 Il principio dice che:
>
> 1. il software dovrebbe essere ”aperto alle estensioni”
> 2. il software dovrebbe essere ”chiuso alle modifiche”

In altre parole, un software progettato bene dovrebbe rendere semplice l'aggiunta di nuove funzionalità (scrivendo nuovo codice), senza che per face ciò sia necessario modificare il codice esistente (chiusura alle modifiche).

> 💡 Si noti che, nella sua enunciazione, il principio OCP non fornice una metodologia esplicita per ottenere gli obiettivi che si propone, ragione per cui potrebbe sembrare inutile. L'operazione chiave per ottenere un progetto aderente al principio e' l'individuazione di quelle parti del software che, con probabilità alta, saranno oggetto di modifica in futuro e l'applicazione a queste parti di opportuni costrutti di astrazione (a volte detti costrutti per ottenere "information hiding"). ESEMPIO: scheda prepagata/fattoria

DIP (Dependecy Inversion Principle)

Nella formulazione di Martin, il principio di inversione delle dipendenze viene enunciato in questo modo:

> 💡 I moduli di alto livello non devono dipendere da quelli di basso livello: **entrambi devono dipendere da astrazioni**. Le astrazioni non devono dipendere dai dettagli; sono i dettagli che dipendono dalle astrazioni.

Le dipendenze "buone" sono quelle che vanno da moduli piu' concreti a moduli piu' astratti, dal basso verso l'alto. Quelle cattive sono quelle che vanno dall'alto verso il basso.

Il principio DIP suggerisce quindi di "invertire" queste dipendenze, sostituendole con altre che invece non creano problemi (perché vanno dal concreto verso l'astratto).

LSP (Liskov Substitution Principle)

Si dice che S e' un sottotipo di T se ad ogni modulo che usa un oggetto t di T e' possibile passare (invece) un oggetto s di S ottenendo comunque un risultato equivalente, cioè un risultato che soddisfa le legittime aspettative dell'utente.

Martin ha riformulato il principio in questi termini:

> 💡 Ogni funzione che usa puntatori o riferimenti a classi base deve essere in grado di usare oggetti delle classi derivate senza saperlo.

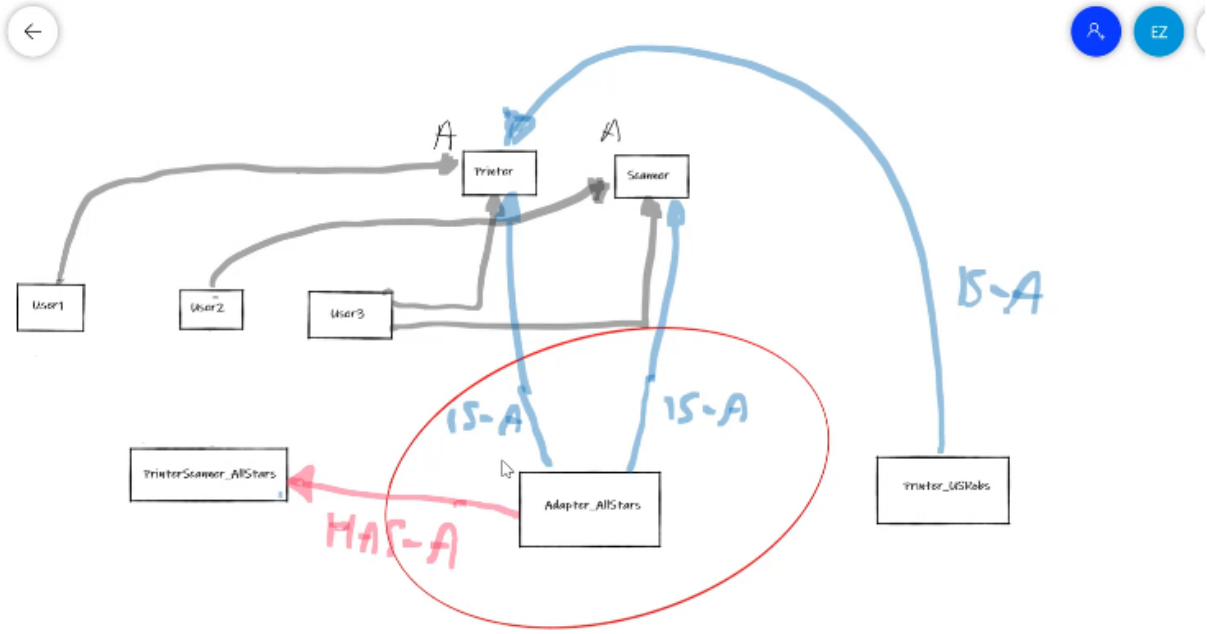
ISP (Interface Segregation Principle)

Il principio di separazione delle interfacce dice che l'utente non dovrebbe essere forzato a dipendere da parti di una interfaccia che non usa. Il progettista dovrebbe preferire tante interfacce "piccole" (thin interfaces) rispetto a poche interfacce "grandi" (fat interfaces).

Aderendo a questo principio, si ottengono i seguenti benefici:

1. l'implementamento può implementare separatamente le interfacce piccole, evitando che un errore su una di queste si propaghi sulle altre;
2. l'implementatore può decidere di implementare solo alcune delle interfacce piccole ottenute dalla separazione dell'interfaccia grande;
3. se una delle interfacce piccole dovesse cambiare, l'utente che non la usa (perché usa le altre) non ne e' minimamente influenzato; al contrario, adottando una sola interfaccia grande, una modifica su una sua parte influenza anche gli utenti che NON fanno alcun uso di quella parte.

Usare adattatore per diverse interfacce




```
#include "Printer.hh"
#include "Printer_USRobs.hh"
```

```
class Adapter_Printer_USRobs : public Printer{
private:
    Printer_USRobs usr;
public:
    Adapter_Printer_USRobs();

    void printer_function_1() override{
        usr.printer_usr_fn1();
    }
    void printer_function_2(const char* str) override{
        usr.printer_usr_fn2(str);
    }
    void printer_function_3(int i) override{
        usr.printer_usr_fn3(i);
    }
};
```

Alcune questioni tecniche sul polimorfismo dinamico

Nel polimorfismo dinamico le classi astratte sono tipicamente formate da metodi virtuali puri, piu' il distruttore della classe che e' dichiarato virtuale, ma non puro. In alcuni casi e' però necessario complicare il progetto (ad esempio, usando l'ereditarietà multipla): quando lo si fa, si corre il rischio di incorrere in errori ed e' opportuno cercare le risposte ad alcune domande tecniche sul polimorfismo dinamico che possono diventare rilevanti quando viene utilizzato al di fuori dei confini stabiliti nei nostri semplici esempi.

1. Ci sono metodo che NON possono essere dichiarati virtuali?

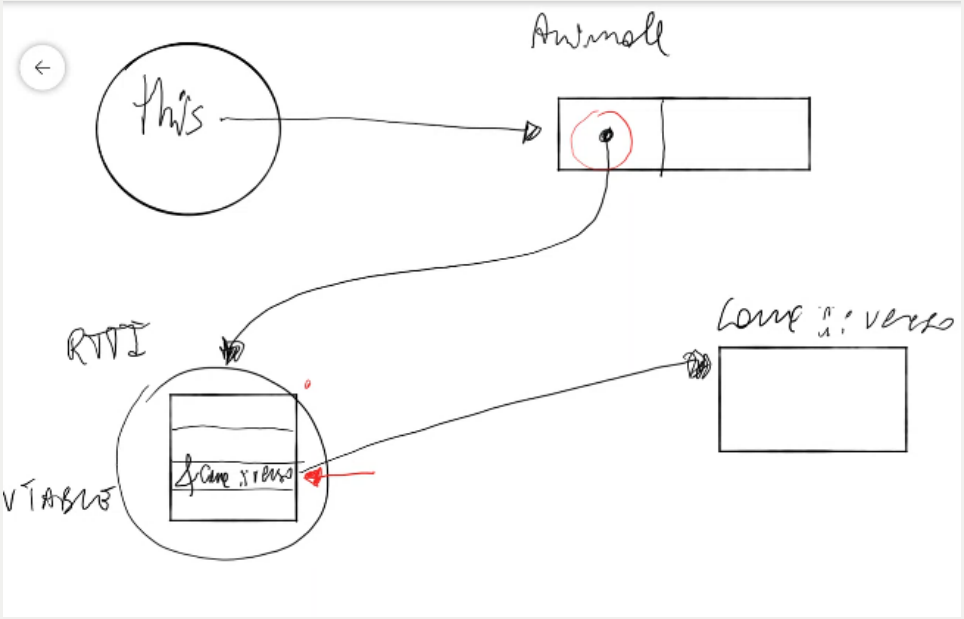
   In particolare, cosa si può dire sulla possibilità o meno di rendere virtuali le seguenti categorie di metodi:

   - costruttori: non può essere dichiarato virtuale
   - distruttori: **deve** essere dichiarato virtuale in una classe virtuale
   - funzioni membro (di istanza, cioè non statiche): può essere virtuale in quanto funzione membro di istanza non statica
   - funzioni membro statiche: non può essere virtuale in quanto a tempo di esecuzione non può essere interrogata

Differenza tra funzione membro di istanza e funzione membro statica

La funzione membro statica si applica alla classe, ovvero non si applica ad un oggetto specifico della classe quindi non ha l'argomento implicito **this.**

Con il puntatore **this** si punta alla classe base dove abbiamo la Virtual Table che contiene tutti i metodi virtuali.



- template di funzione membro (non statiche): non possono essere virtuali per motivi di efficienza

I template di funzione sono strumenti per generare un numero a piacere di funzioni

- funzioni membro (non statiche) di classi templatiche: può essere virtuale ma non e' obbligatorio

2. Come faccio a costruire una copia di un oggetto concreto quando questo mi viene fornito come puntatore/riferimento alla classe base?

```
class Animale{
    virtual Animale* copia() const=0;
};

class Cane : public Animale{
    // "costruttori virtuali"
    Cane* copia() const override{
        return new Cane(*this);
    }
};

void foo(const Animale* pa){
    // crea una copia
    Animale* pa2 = pa ->copia();

    // auto pa2=pa; auto=const Animale*
    // Animale a2=*pa; non si puo' fare 1: se dovesse compila
                                        // un object slicing una

}
```

💡 Con la classe derivata posso rafforzare le post condizioni, cosa che non si può fare con la classe base

3. Cosa succede se si invoca un metodo virtuale durante la fase di costruzione o di distruzione di un oggetto?

   Costruzione:

📖 Si causa un undefined behaviour, in quanto il dato non e' stato costruito e quindi non inizializzato nel caso di un costruttore.

NOTA

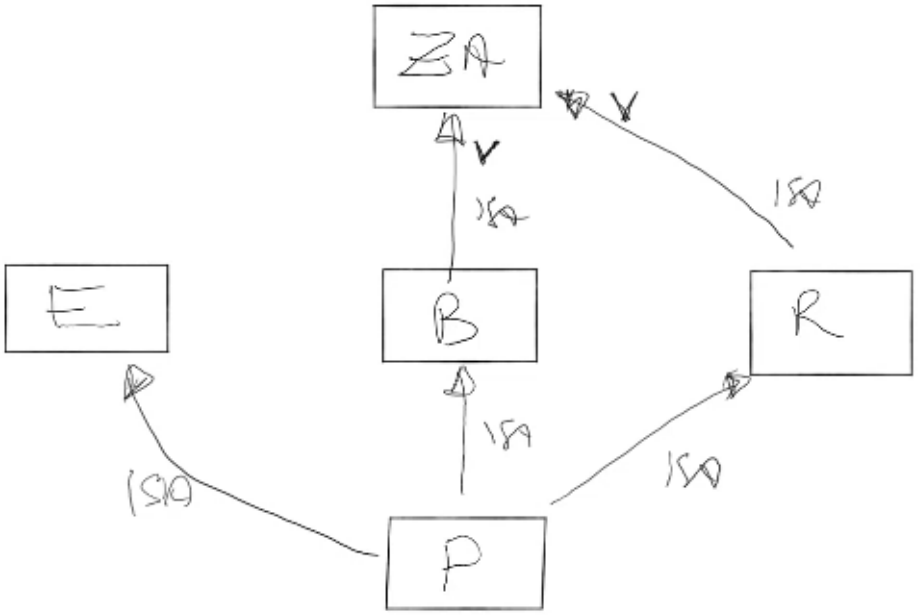
Dallo standard C++11 non abbiamo piu' questo problema

Distruzione:

> 📖 Si causa un undefined behaviour, in quanto il dato prossimo alla distruzione punterà al dato appena distrutto.
>
> **NOTA**
>
> Dallo standard C++11 non abbiamo piu' questo problema

4. Come funziona l'ereditarietà multipla quando NON ci si limita al caso delle interfacce astratte?
   - scope e ambiguità
   - classi base ripetute
   - classi base virtuali
   - semantica speciale di inizializzazione

> 💡 **regola speciale**
>
> Prima faccio la costruzione/distruzione evitando di costruire/distruggere le classi base
> Per fare ciò devo ereditare con
> `virtual`
> es:
>
> ```
> class A : virtual public B{...};
> ```



> 💡 Nella semantica speciale di inizializzazione le classi virtuali vengono inizializzate per prime, indipendentemente dalla loro posizione di inizializzazione

5. Quali sono gli usi del polimorfismo dinamico nella libreria standard?
   - classi eccezione standard
   - classi iostream

Fine