

# Hello World

---

## A cosa serve questo programma?

Lo scopo di questo programma, in realtà, è quello di effettuare un test sulla corretta installazione dell'ambiente di sviluppo scelto per il C++. Ovvero, ci si vuole assicurare di avere un compilatore funzionante, le librerie di sistema correttamente installate, ecc.

Dal nostro punto di vista, comunque, questo semplice programma si presta bene ad evidenziare la differenza che esiste tra una conoscenza superficiale del linguaggio C++ ed una conoscenza un po' più approfondita.

### Esempi di domande da esame:

- perché la funzione `main` è dichiarata per restituire un valore intero?
- perché non esiste la corrispondente istruzione di `return`?
- perché devo qualificare il nome (`cout`) del canale di output con `std`?
- che cosa indica `std`?
- qual è la differenza tra queste due varianti? `std::cout << "Hello, world!" << std::endl;` vs `std::cout << "Hello, world!\n";`
- che cosa è `iostream`?
- perché devo includere `iostream`?
- perché devo usare le parentesi angolate (e non le virgolette) quando includo `iostream`?
- a cosa si riferiscono le due occorrenze dell'operatore infisso `<<`?
- sono invocazioni di operatori built-in o si tratta di funzioni definite dall'utente?

Esempio di helloworld:

```
#include  
  
int main() {  
    std::cout << "Hello, world!" << std::endl;  
}
```

[Torna all'indice](#)

---

## Processo di compilazione

Il compilatore `g++` è un wrapper per il compilatore `gcc` (*Gnu Compiler Collection*), il quale è in realtà una collezione di compilatori per diversi linguaggi.

In senso lato, il processo di compilazione prende in input file sorgente e/o librerie e produce in output file eseguibili o librerie. Esso segue alcuni passaggi fondamentali:

1. Il **preprocessore** elabora il codice del file sorgente per produrre una *unità di traduzione*.
2. Il **compilatore** (in senso stretto) elabora l'unità e produce codice *assembler*.
3. L'**assemblatore** produce da questo il file oggetto.
4. Il **linker** si occupa infine di realizzare i collegamenti tra i vari file oggetto e le librerie al fine di ottenere l'eseguibile (o una libreria).

## Comandi step-by-step

1. `g++ -E hello.cpp -o hello.preproc.cpp` L'opzione `-E` indica al compilatore di fermarsi subito dopo la fase di preprocessing. Le prime 10 righe dell'output sono le seguenti:

```
# 0 "hello.cpp"  
# 0 ""  
# 0 ""
```

```
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "" 2
# 1 "hello.cpp"
# 1 "/usr/include/c++/12.2.1/iostream" 1 3
# 36 "/usr/include/c++/12.2.1/iostream" 3
# 37 "/usr/include/c++/12.2.1/iostream" 3
```

2. g++ -Wall -Wextra -S hello.cc -o hello.s L'opzione -S permette di fermarsi alla fase di compilazione in senso stretto, con la generazione del codice assembler.

```
.file "hello.cpp"
.text
.local _ZStL8__ioinit
.comm _ZStL8__ioinit,1,1
.section .rodata
.LC0:
.string "Hello, world!"
.text
.globl main
.type main, @function
main:
.LFB1761:
.cfi_startproc
pushq %rbp
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_offset 6, -16
leaq .LC0(%rip), %rax
movq %rax, %rsi
leaq _ZSt4cout(%rip), %rax
movq %rax, %rdi
```

3. g++ -Wall -Wextra -c hello.cpp -o hello.o L'opzione -c produce il codice oggetto (scritto in linguaggio macchina) fermandosi prima del collegamento.

4. g++ -Wall -Wextra hello.cc -o hello Quando **NON** si specifica nessuna delle opzioni -E, -S, -c, il compilatore termina dopo avere effettuato il collegamento, utilizzando il linker (ld), producendo il file eseguibile hello (senza suffisso, come consuetudine per l'ambiente Linux). In questo caso, avendo un solo file oggetto, il collegamento avviene tra questo file e i file che formano la libreria standard del \$C\$++ (che vengono coinvolti implicitamente, senza doverli specificare come argomenti per il compilatore). In altri casi, si può indicare il percorso degli headers da includere con l'opzione -I. Esempio:

[Torna all'indice](#)

---

## Considerazioni sul contenuto dei files prodotti

Anzitutto le dimensioni sono molto variabili:

```
$ du -b hello*
16008 hello
81     hello.cpp
2736 hello.o
820162 hello.preproc.cpp
2032 hello.s
```

In particolare l'unità di traduzione risulta molto estesa: essa contiene l'espansione dell'unica direttiva di inclusione del sorgente (#include <iostream>). Il preprocessore include dunque il codice di numerosi altri file sorgente che in questo caso fanno parte della libreria standard del \$C\$++. I nomi di questi file (e la loro posizione nel filesystem) si possono ottenere osservando le direttive del preprocessore rimaste nell'unità di traduzione (le linee che iniziano con il carattere #):

```
...
# 1 "/usr/include/c++/9/iostream" 1 3
...
# 1 "/usr/include/x86_64-linux-gnu/c++/9/bits/basic_ios.h" 1 3
brav# 1 "/usr/include/features.h" 1 3 4
...
```

Queste direttive servono al compilatore per generare **messaggi di errore** che facciano riferimento ai nomi dei file e ai numeri di riga dei file sorgenti (e non al numero di riga dell'unità di traduzione, cosa che sarebbe alquanto scomoda per il programmatore).

Si deve notare che gli header files inclusi sono necessari in quanto contengono la **dichiarazione** della variabile `std::cout` (e del corrispondente tipo), dell'operatore di output `operator<<` e del modificatore `std::endl`.

```
namespace std {
// ...
typedef basic_ostream ostream;
// ...
extern ostream cout;
// ...
extern template ostream& endl(ostream&);
// ...
extern template ostream& operator<<(ostream&, const char* );
// ...
}
```

Osserviamo che:

- `std::ostream` è un alias per la classe `basic_ostream<char>` ottenuta instanziando il **template** di classe `basic_ostream` con il tipo `char`.
- Il modificatore `std::endl` è una funzione templatICA specializzata con il tipo `ostream`.
- La prima occorrenza di `operator<<` è una chiamata a una funzione templatICA, specializzata su `ostream` e `const char*`.
- La seconda occorrenza di `operator<<` fa riferimento invece ad un altro tipo di funzione che prende come argomento due parametri di tipo `ostream`.

Perchè non è necessario qualificare l'operatore `<<` all'interno del proprio **namespace** (`std::<<`)? Il \$C\$++ utilizza l'**Argument-Dependent Lookup** (ADL): se viene utilizzato un argomento definito dall'utente come argomento di una funzione (o operatore) e non viene esplicitato il namespace di quest'ultima, si cerca nell'ordine

1. nello scope del chiamante, e
2. in tutti i namespaces a cui appartengono gli argomenti, partendo dal primo a sinistra.

Ad esempio `operator<<(ostream&, const char*)` prende in input un tipo `ostream&` dichiarato all'interno del namespace `std`; in esso si trova anche la definizione dell'operatore da utilizzare.

## Esempio: ADL

```
#include
#include

namespace A {
    class Message : public std::string {
public:
    Message() : std::string("This is a message.") {};
};

std::ostream& operator<<(std::ostream& os1, std::string& msg) {
    std::cout << "Operator << in namespace A" << std::endl;
    return os1;
} // ! NAMESPACE_A

int main() {
    A::Message msg;
    std::string str("Hello World!");
    std::cout << str << std::endl; // std::operator<<(std::ostream&, std::string&
    std::cout << msg; // A::operator<<(std::ostream&, std::string&
}
```

[Torna all'indice](#)

# Ricapitolando

Giunti al termine di questa discussione su "Hello, world!" è forse il caso di ricapitolare alcuni concetti chiave.

1. Tecnicamente, quando si dice che un programma è formato da un solo file sorgente, si commette un errore; come abbiamo visto, anche un programma banale come "Hello, world!" necessita di oltre un centinaio di header file di sistema (il numero esatto dipende dall'implementazione specifica usata), oltre a `helloworld.cpp`.
2. I programmi "veri" sono formati da numerosi file sorgente, anche quando si escludono gli header file di sistema; è necessario comprendere i meccanismi che consentono di separare il programma in diversi file sorgente, per poi compilarli separatamente e infine collegarli in modo corretto; occorre inoltre imparare come compilare e collegare correttamente un programma che dipenda da librerie software di terze parti (ovvero, diverse dalla libreria standard del C++).
3. In generale, ogni volta che si scrive qualche linea di codice è necessario chiedersi qual è la sua funzione, evitando di dare la risposta: "Non ne ho idea, so solo che si è sempre fatto così". Questa è la risposta di un programmatore che NON sa risolvere eventuali problemi tecnici che si dovessero presentare e deve per forza chiedere consiglio a programmatore più esperti. La nostra ambizione dovrebbe essere quella di (iniziare il lungo cammino per) diventare noi stessi programmatore esperti e sapere come affrontare i problemi tecnici legati alla programmazione. Si noti che queste competenze sono ben separate ed indipendenti rispetto alla conoscenza dello specifico dominio applicativo per il quale si è deciso di sviluppare uno strumento software.

[Torna all'indice](#)

# Dichiarazioni e Definizioni

---

## Dichiarazione vs Definizione

Una **DICHIARAZIONE** è un costrutto del linguaggio che introduce (dichiara) un nome per una entità (tipo di dato, variabile, funzione, template di classe, template di funzione, ecc.).

Una **DEFINIZIONE** è una dichiarazione che, oltre al nome, fornisce ulteriori elementi per caratterizzare l'entità (per esempio, la struttura interna di un tipo di dato, l'implementazione del corpo di una funzione, ecc.).

Le dichiarazioni che NON sono anche definizioni vengono talvolta chiamate **dichiarazioni "forward"** (perché rimandano la definizione ad un momento successivo) o anche dichiarazioni **"pure"**. Vediamo la differenza tra dichiarazioni pure e definizioni per varie tipologie di entità.

[Torna all'indice](#)

---

## Tipi di dato

Dichiarazione pura del tipo `s`:

```
struct S;
```

Definizione del tipo `T`:

```
struct T { int a; };
```

Nel primo caso, non conosco la struttura del tipo `s` e, di conseguenza, non posso creare oggetti del tipo `s` (per esempio, il compilatore non saprebbe quanta memoria allocare per memorizzare un tale oggetto).

Nel secondo caso, siccome `T` è definita, conosco la struttura e posso creare oggetti di tipo `T`.

Ci si potrebbe chiedere quale sia l'utilità di avere una dichiarazione pura di tipo, visto che non si possono creare valori di quel tipo.

In realtà sono utili quando occorre definire puntatori o riferimenti a valori del tipo `T`, senza dover conoscere il tipo `T`; si parla in questo caso di **puntatori opachi**:

```
struct T;  
T* t_ptr; // un puntatore "opaco" a T
```

Ci si potrebbe anche chiedere perché il linguaggio insista, nel caso di dichiarazioni pure, a richiedere che il programmatore dichiari se il nome introdotto è un tipo o un valore (cioè, perché occorre indicare la parola `struct / class` per dire che è un tipo).

La risposta è che questa informazione è essenziale per poter fare il *parsing* (cioè l'analisi sintattica) del codice. Consideriamo infatti una variante dell'esempio precedente:

```
nome1 * nome2
```

Il compilatore che si trova di fronte a questo codice lo può interpretare in due modi completamente differenti:

- `nome2` è dichiarato essere un puntatore al tipo `nome1`, oppure
- si richiede di applicare l'operatore binario `*` ai due valori `nome1` e `nome2`

Se il compilatore vede che `nome1` è un *tipo*, sceglie la prima opzione; se vede che è invece un *valore*, sceglie la seconda.

[Torna all'indice](#)

## Un caso speciale

Nel caso del \$C\$++ 2011 (o successivo), è possibile anche fornire una dichiarazione pura per un tipo *enumerazione enum*, cosa che non era possibile fare con il \$C\$++ 2003:

```
enum E : int;           // dichiarazione pura
enum E : int { a, b, c, d}; // definizione
```

[Torna all'indice](#)

---

## Variabili

Dichiarazione pura di variabile (globale):

La parola chiave `extern` vuole dire: Non creare lo spazio per inizializzare la variabile, lo farà qualcun altro

```
extern int a;
```

Definizione di variabile:

```
int b;
int c = 1;
extern int d = 2; // definizione, perchè viene inizializzata
```

Nel caso della dichiarazione pura, il compilatore viene informato dell'esistenza di una variabile di nome `a` e di tipo `int`, ma la creazione di tale variabile verrà effettuata altrove (probabilmente in un'altra unità di traduzione).

Nel caso delle definizioni, invece, il compilatore si occuperà di creare le variabili `b`, `c`, `d` e, se richiesto di inizializzarle.

[Torna all'indice](#)

---

## Funzioni

Come identifico una funzione (in ordine di importanza)?

1. Identifco il namespace.
2. Identifco il nome.
3. Identifco il numero e tipo di argomenti.
4. Come sono stanziali i parametri del template.

Il tipo di ritorno non serve ad identificare una funzione.

Dichiarazioni pure di funzioni:

```
void foo(int a);
//poco usato
extern void foo(int a);
// Sono due ri-dichiarazioni della stessa funzione
```

Definizione di funzione:

```
void foo(int a) {
    std::cout << a;
}
```

La parola chiave `extern` è opzionale e, in pratica, è usata raramente: *le definizioni hanno il corpo*.

Possiamo anche interfacciarsi al mondo interno cambiando le regole di collegamento; in questo caso la funzione `foo` utilizzerà i metodi del \$C\$ e non \$C\$++:

```
extern "C" void foo(int a);
```

[Torna all'indice](#)

---

## Template (di classe e di funzione)

I template di classe non sono classi; essi infatti si possono considerare come uno schema, che indica al compilatore come dichiarare e/o definire ciò che segue attorno a dei *tipi parametrici*.

Dichiarazione pura di template di classe:

```
template struct S;
```

Definizione di template di classe:

```
template
struct S {
    T t;
};
```

Dichiarazione pura di template di funzione:

```
template
T add(T t1, T t2);
```

Definizione di template di funzione:

```
template
T add(T t1, T t2) {
    return t1 + t2;
}
// Il compilatore non sa cosa fare con l'operatore '+'.
// Lo saprà solo quando scoprirà di che tipo saranno le variabili 't1' e 't2'.
```

[Torna all'indice](#)

# Scope (campo d'azione)

---

## Entità

Ogni [dichiarazione](#) presente in una unità di traduzione introduce un nome per una **entità**.

Tale nome può essere utilizzato solo in alcuni punti dell'unità di traduzione: le porzioni di codice in cui il nome è "visibile" sono dette essere il *campo di azione* (in inglese, **scope**) per quel nome. L'ampiezza dello scope per un nome varia a seconda della tipologia di dichiarazione e del contesto in cui questa appare. Si distinguono diverse tipologie di scope.

[Torna all'indice](#)

---

## Scope di Namespace (incluso lo scope globale)

I [namespaces](#) sono utilizzati per organizzare il codice in gruppi logici, allo scopo di evitare conflitti di nomi e migliorare la leggibilità del codice. Una dichiarazione che non è racchiusa all'interno di una `struct/class` e/o all'interno di una funzione ha scope di `namespace`.

Si noti che lo scope globale è anche esso uno scope di `namespace` (al quale ci si può riferire usando il qualificatore di scope `::`).

Il nome è visibile, all'interno di quel `namespace`, a partire dal punto di dichiarazione e fino al termine dell'unità di traduzione (in particolare, NON è visibile prima del punto di dichiarazione).

In sostanza, questo è il motivo per il quale le inclusioni degli header file sono collocate all'inizio dei file sorgente.

```
namespace N {
    void foo() {
        // ERRORE: `bar` e `a` non sono visibili in questo punto
        // (vengono dichiarate dopo)
        bar(a);
    }

    int a; // definizione di 'a'
    void bar(int n) {
        a += n;
        // OK: 'a' è visibile in questo punto (dichiarata prima)
        // della funzione 'bar'
    }
} // ! namespace N
```

[Torna all'indice](#)

---

## Scope di Blocco

Un nome dichiarato in un *blocco* (porzione di codice all'interno del corpo di una funzione racchiusa tra parentesi graffe) è locale a quel blocco.

Anche in questo caso, la visibilità inizia dal punto di dichiarazione e termina alla fine del blocco.

```
void foo() {
    // ...
    { // inizio blocco

        // ...
        int j /* inizio scope di blocco per j */ = expr;
        // ...
        std::cout << j;
        // ...
    }
}
```

```
} // fine dello scope di blocco per j
```

Vi sono alcune regole speciali per i costrutti `for`, `while`, `if`, `switch` e per i blocchi `try/catch`:

```
for (int i = 0; i != 10; ++i) {
    // i ha lo scope del blocco for
}

if (T* ptr = foo()) {
    // ptr è visibile qui (e vale ptr != nullptr)
} else {
    // ptr è visibile anche qui (e vale ptr == nullptr)
}
```

E se avessimo dichiarato `ptr` fuori dal costrutto `if`? Avrebbe portato ad una estensione dello scope del puntatore, aumentando la possibilità di errori e/o comportamenti non voluti dal programmatore.

```
switch (int c = bar()) {
    case 0:
        break;

    case 1:
        do_something(c); // c è visibile qui
        break;

    case 2:
        do_something_different();
        break;

    default:
        std::cerr << "unexpected value c = " << c;
        break;
}

try {
    int a = 5;
    // ...
}
catch (const std::string& s) {
    std::cerr << s; // 's' è visibile qui
    // ATTENZIONE: 'a' NON è visibile qui
}
```

[Torna all'indice](#)

---

## Scope di Classe

Qual è la differenza tra `struct` e `class`? Che le classi garantiscono l<sup>\*\*\*</sup>information hiding<sup>\*\*</sup> (l'utente può gestire metodi e attributi con `public` e `private`), mentre le `struct` no. Per le `struct` la visibilità di default è `*public*` mentre nelle classi è `*private*`.

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 dopo
    }

    int a;
    void bar(int n) { a += n; }
};
```

I membri di una classe posso essere acceduti dall'esterno della classe nei modi seguenti:

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

I membri di una classe `s` possono essere acceduti anche da classi che sono derivate (anche indirettamente) dalla classe `s` (in quanto sono ereditati dalle classi derivate). In caso di *overloading* di metodi si può accedere a quelli della classe base usando il risolutore di scope `:::`.

[Torna all'indice](#)

---

## Scope di Funzione

Le etichette (*label*) di destinazione delle istruzioni `goto` hanno scope di funzione: sono visibili in tutta la funzione che le racchiude, indipendentemente dai blocchi.

```
void foo() {
    int i;
    {
        inizio: // visibile anche fuori dal blocco

        i = 1;
        while (true) {
            // ...
            if (condizione)
                goto fine; // fine è visibile anche se dichiarata dopo
        }
    }

    fine:

    if (i > 100)
        goto inizio;
    return i;
}
```

L'uso dei `goto` e delle etichette è considerato cattivo stile e andrebbe limitato ai casi (pochissimi) in cui risultano essenziali.

[Torna all'indice](#)

---

## Scope delle costanti di enumerazione: un caso speciale.

Le costanti di enumerazione dichiarate secondo lo stile \$C\$++ 2003.

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

Hanno come scope quello del corrispondente tipo enumerazione `Colors` (ovvero, sono visibili "fuori" dalle graffe che le racchiudono).

Questo può causare problemi di conflitto di nomi:

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

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

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;
}
```

Il cast è necessario perché le `enum class` impediscono anche le conversioni implicite di tipo verso gli interi.

# Riduzioni ed estensioni dello scope di una dichiarazione

Quello introdotto precedentemente è il cosiddetto scope potenziale di una dichiarazione. Lo scope potenziale può essere modificato da alcuni costrutti del linguaggio.

## Hiding di un nome

Quando si annidano campi di azione, è possibile che una dichiarazione nello scope interno nasconde un'altra dichiarazione (con lo stesso nome) dello scope esterno.

Si parla di **hiding di un nome**.

```
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
}
```

Si può avere hiding anche per i membri ereditati da una classe, perché lo scope della classe derivata è considerato essere incluso nello scope della classe base:

```
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()
};
```

[Torna all'indice](#)

## Estensioni della visibilità di un nome

Per accedere ad un nome dichiarato in uno scope differente, è spesso possibile utilizzare la versione qualificata del nome. Per esempio, dentro la classe `Derived` vista sopra, si può accedere ai dati e ai metodi della classe `Base` scrivendo `Base::a` e `Base::foo`. Lo stesso dicasi nel caso dello scope di namespace (si ricordi l'uso di `std::cout`).

Se però un nome deve essere utilizzato molto spesso in una posizione in cui non è visibile senza qualificazione, può essere scomodo doverlo qualificare in ogni suo singolo uso.

Per evitare ciò, si possono usare le **dichiarazioni di using** (*using declaration*):

```
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 dichiarazione di `using` rende disponibile (nel contesto in cui viene inserita) il nome riferito, che da lì in poi potrà essere usato senza qualificazione. Chiaramente, nel caso di un nome di tipo o di una variabile, è necessario che nello stesso contesto *NON* sia già presente un'altra entità con lo stesso nome.

```
void foo() {
    int cout = 5;
    using std::cout; // error: 'cout' is already declared in this scope
}
```

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 {
    // rendo visibili in questo scope tutti i metodi
    // di nome "foo" presenti in Base
    using Base::foo;

    // foo(double) va in overloading con foo(int) e foo(float)
    void foo(double d);
};
```

[Torna all'indice](#)

---

## Direttive using

Cosa ben distinta rispetto alle dichiarazioni di `using` sono le **direttive di `using`** (*using directive*). La sintassi è la seguente:

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

La direttiva di `using` *NON* introduce dichiarazioni nel punto in cui viene usata; piuttosto, aggiunge il namespace indicato tra gli scope nei quali è possibile cercare un nome per il quale *NON* si trovino dichiarazioni nello scope corrente.

Per capire la differenza, consideriamo l'esempio seguente:

```
#include

void foo() {
    int endl = 42;
    using namespace std;
    cout << "Hello" << endl;
}
```

Vedendo l'uso del nome `cout`, il compilatore lo cerca nello scope corrente (il blocco della funzione). Non trovandolo, continua la ricerca negli scope che racchiudono la funzione `foo` e, grazie alla direttiva di `using`, anche nello scope del `namespace std` (trovandolo).

Vedendo l'uso del nome `endl`, il compilatore lo cerca nello scope corrente e trova la dichiarazione della variabile intera, completando la ricerca. La direttiva di `using` in questo caso *NON* entra in gioco e la funzione stamperà la stringa "Hello42" (senza andare a capo).

[Torna all'indice](#)

# Tempo di vita (lifetime)

---

Abbiamo visto come il concetto di scope consenta di stabilire in quali punti del codice (*dove*) è visibile il nome introdotto da una dichiarazione. Oltre alla dimensione spaziale, è necessario prendere in considerazione anche la dimensione temporale, per stabilire in quali momenti (*quando*) è legittimo interagire con determinate entità.

Alcune entità (tipi di dato, funzioni, etichette) possono essere riferite in qualunque momento durante l'esecuzione del programma.

Un oggetto memorizzato in memoria, invece, è utilizzabile solo dopo che è stato creato e soltanto fino a quando viene distrutto, ovvero ogni accesso è valido solo durante il suo tempo di vita (lifetime).

---

## Note iniziali

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.

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

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 di una espressione* che crea implicitamente un nuovo oggetto (oggetto temporaneo, senza nome).

Il tempo di vita di un oggetto:

1. *inizializzazione*, che è composta da due fasi distinte:
  - allocazione della memoria "grezza";
  - inizializzazione della memoria (quando prevista);
2. *deallocazione*, che è anch'essa composta da due fasi distinte:
  - invocazione del distruttore (quando previsto);
  - deallocazione della memoria "grezza".

Si noti che un oggetto la cui costruzione NON termina con successo, NON avendo iniziato il suo tempo di vita, NON dovrà terminarlo, ovvero per quell'oggetto NON verrà eseguita la distruzione.

Si noti anche che DURANTE le fasi di creazione e di distruzione di un oggetto si è fuori dal suo tempo di vita, per cui le operazioni che è possibile effettuare sull'oggetto sono molto limitate (le regole del linguaggio sono complicate e comprendono numerosi casi particolari, che per il momento non è opportuno approfondire).

[Torna all'indice](#)

---

## Storage duration ("Allocazione")

Vi sono diversi tipi di *storage duration* (aka *allocazione*) per gli oggetti in memoria:

1. [Allocazione statica](#)
2. [Allocazione thread local](#)
3. [Allocazione automatica](#)
4. [Allocazione automatica di temporanei](#)
5. [Allocazione dinamica](#)

[Torna all'indice](#)

---

## Allocazione statica

La memoria di un oggetto ad allocazione statica dura per tutta l'esecuzione del programma.

Sono dotate di memoria ad allocazione statica:

1. **Le variabili definite a namespace scope (dette globali).** Queste sono create e inizializzate *prima* di iniziare l'esecuzione della funzione main, nell'ordine in cui compaiono nell'unità di traduzione in cui sono definite.

Nel caso di variabili globali definite in diverse unità di traduzione, l'ordine di inizializzazione *NON* è specificato.

```
// Il seguente programma stampa due stringhe,  
// anche se il corpo della funzione main è vuoto  
  
#include  
  
struct S {  
    S() { std::cout << "costruzione" << std::endl; }  
    ~S() { std::cout << "distruzione" << std::endl; }  
};  
  
S s; // allocazione globale  
  
int main() {}
```

2. **I dati membro di classi dichiarati usando la parola 'static'.** Questi sono creati come le variabili globali del punto precedente.
3. **Le variabili locali dichiarate usando la parola chiave 'static'.** Queste sono allocate prima di iniziare l'esecuzione della funzione main, ma sono inizializzate (solo) la prima volta in cui il controllo di esecuzione incontra la corrispondente definizione (nelle esecuzioni successive l'inizializzazione non viene eseguita).

```
// Il seguente programma stampa una stringha,  
// indicando il numero totale di volte in cui  
// la funzione `foo()` è stata chiamata:  
  
#include  
  
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();  
    }  
}
```

[Torna all'indice](#)

---

## Allocazione thread local

Un oggetto thread local è simile ad un oggetto globale, ma il suo ciclo di vita non è collegato al programma, bensì ad ogni singolo thread di esecuzione creato dal programma (esiste una istanza distinta della variabile per ogni thread creato). Il supporto per il multithreading è stato introdotto con lo standard C++ 2011.

[Torna all'indice](#)

---

## Allocazione automatica

Una variabile locale ad un blocco di funzione è dotata di allocazione automatica: l'oggetto viene creato dinamicamente (sullo *stack*) ogni volta che il controllo entra nel blocco in cui si trova la dichiarazione e viene automaticamente distrutto (rimuovendolo dallo *stack*) ogni volta che il controllo esce da quel blocco.

```
void foo() {
    int a = 5;
{
    int b = 7;
    std::cout << a + b;
} // 'b' viene distrutta automaticamente all'uscita da questo blocco
std::cout << a;
} // 'a' viene distrutta automaticamente all'uscita da questo blocco
```

Nel caso di funzioni ricorsive, sullo *stack* possono esistere contemporaneamente più istanze distinte della stessa variabile locale.

[Torna all'indice](#)

---

## Allocazione automatica di temporanei

L'allocazione automatica di temporanei avviene quando un oggetto viene creato per memorizzare il valore calcolato da una sottoespressione 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";
}
```

L'oggetto temporaneo viene *distrutto* quando termina la valutazione dell'espressione completa che contiene lessicalmente il punto di creazione. Nell'esempio precedente, il temporaneo è distrutto al termine dell'esecuzione di `foo`, ma prima della stampa di "fine".

Il lifetime di un oggetto temporaneo può essere esteso se l'oggetto viene utilizzato per inizializzare un riferimento; in tale caso, l'oggetto verrà distrutto nel momento in cui verrà distrutto il riferimento.

Per esempio:

```
void bar2() {
    // il temporaneo S(42) è usato per inizializzare il riferimento s
    const S& s = S(42);
    std::cout << "fine";

    // il temporaneo è distrutto quando si esce dal blocco,
    // dopo avere stampato "fine"
}
```

[Torna all'indice](#)

---

## Allocazione dinamica

Un oggetto (senza nome) può essere allocato dinamicamente nella memoria heap usando l'espressione `new` (che restituisce l'indirizzo dell'oggetto allocato, che va salvato in un opportuno puntatore).

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

**La distruzione dell'oggetto NON è automatica**, ma viene effettuata sotto la responsabilità del programmatore utilizzando l'istruzione `delete` (sul puntatore che contiene l'indirizzo dell'oggetto).

```
delete pi;  
// l'oggetto puntato da pi è stato distrutto,  
// ma pi continua a contenere il suo indirizzo, non più valido;  
// pi è diventato un "dangling pointer" (puntatore penzolante)
```

L'allocazione dinamica è una sorgente inesauribile di errori di programmazione:

- **Errore "use after free"**: Usare (per leggere o scrivere sull'oggetto puntato) un puntatore dangling. In pratica si usa un oggetto dopo che il suo lifetime è concluso.
- **Errore "double free"**: Usare la delete due o più volte sullo stesso indirizzo, causando la distruzione di una porzione di memoria che non era più allocata (o era stata riutilizzata per altro).
- **"Memory leak"**: Si distrugge l'unico puntatore che contiene l'indirizzo dell'oggetto allocato prima di avere effettuato la delete (quindi l'oggetto non verrà mai più distrutto, causando come minimo uno spreco di memoria).
- **Accesso ad un "wild pointer"**: Variante della use after free; si segue un puntatore che indirizza memoria "causale", leggendo o scrivendo dove non c'è un oggetto (o non c'è l'oggetto inteso).
- **Accesso al "null pointer"**: Si prova ad accedere ad un puntatore nullo.

[Torna all'indice](#)

# Tipi, qualificatori, costanti letterali

---

## I tipi fondamentali (non strutturati)

- Booleani: `bool`
- Carattere:
  1. *narrow character type*: `char`, `signed char`, `unsigned char`
  2. *wide character type*: `wchar_t`, `char16_t`, `char32_t`
- Interi standard con segno: `signed char`, `short`, `int`, `long`, `long long`
- 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 soggetti a promozioni di tipo.

- Floating point: `float`, `double`, `long double`
- Void: ha un insieme vuoto di valori; serve per indicare che una funzione non ritorna alcun valore o, usando un cast esplicito, che il valore di una espressione deve essere scartato.

```
(void) foo(3); // chiama foo(3) e scarta il risultato prodotto
```

- `std::nullptr_t`: è un tipo puntatore convertibile implicitamente in qualunque altro tipo puntatore; ha un solo valore possibile, la costante letterale `nullptr`, che indica il puntatore nullo (non dereferenziabile).

[Torna all'indice](#)

---

## I tipi composti

- Riferimenti a lvalue: `T&`
- Riferimenti a rvalue: `T&&`
- Puntatori: `T*`
- Tipi array: `T[n]`
- Tipi funzione: `T(T1, ..., Tn)`
- Enumerazioni e class / struct

[Torna all'indice](#)

---

## Qualificatore const

I tipi elencati sopra sono detti non qualificati. Nel linguaggio C++ esistono i qualificatori `const` e `volatile`. Nel discorso che segue consideriamo solo il qualificatore `const`, il cui uso è essenziale per una corretta progettazione e uso delle interfacce software e come strumento di ausilio alla manutenzione del software.

Dato un tipo `T`, è possibile fornirne la versione qualificata `const T`. L'accesso ad un oggetto (o una parte di un oggetto) attraverso una variabile il cui tipo è dotato del qualificatore `const` è consentito in sola lettura (cioè, non sono consentite le modifiche).

Si noti che nel caso di tipi composti è necessario distinguere tra la qualificazione del tipo composto rispetto alla qualificazione delle sue componenti.

Per esempio:

```
struct S {  
    int v;  
    const int c;  
    S(int cc) : c(cc) { v = 10; } // lista di inizializzazione delle classi basi e dei dati membro  
};
```

```

int main() {
    const S sc(5)
    sc.v = 20; // errore: 'sc' è const e anche le sue componenti
    S s(5);
    s.v = 20; // legittimo: 's' non è const e 'S::v' non è const
    s.c = 20; // errore: 's' non è const, ma 'S::c' è const
}

```

Si noti inoltre che lo stesso oggetto può essere modificabile o meno a seconda del percorso usato per accedervi:

```

struct S { int v; };

void foo() {
    S s;
    s.v = 10; // legittimo
    const S& sr = s; // riferimento a 's', qualificato const
    sr.v = 10; // errore: non posso modificare 's' passando da 'sr'.
}

```

[Torna all'indice](#)

---

## Costanti letterali

Il linguaggio mette a disposizione varie sintassi per definire valori costanti; a seconda della sintassi usata, al valore viene associato un tipo specifico, che in alcuni casi dipende dall'implementazione.

- 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

[Torna all'indice](#)

---

## Note varie

- L'elenco NON è esaustivo; serve a dare un'idea delle potenziali complicazioni. Per esempio, nel caso degli interi consideriamo solo la sintassi decimale, ma esistono anche:
  - la sintassi binaria (0b1100, che rappresenta il numero decimale 12)
    - la sintassi ottale (014, che rappresenta il numero decimale 12)
    - la sintassi esadecimale (0xC, che rappresenta il numero decimale 12)
- In assenza di suffissi (U, L, LL) ad una costante *decimale* intera viene attribuito il *primo* tipo tra int, long e long long che sia in grado di rappresentarne il valore. Il tipo dipende quindi dalla particolare implementazione utilizzata: ad un valore molto grande può essere assegnato il tipo long o long long.
- Le regole per le altre sintassi (booleana, ottale, esadecimale) prendono in considerazione anche i tipi unsigned.
- I suffissi delle costanti intere e floating point sono case insensitive; le convenzioni di solito privilegiano la versione maiuscola (raramente la minuscola) per maggiore leggibilità.

[Torna all'indice](#)

---

## Altre costanti letterali

In presenza del suffisso `U`, si sceglie la variante `unsigned`. In presenza del suffisso `L`, l'ampiezza è scelta tra `long` e `long long`. In presenza del suffisso `LL`, l'ampiezza è `long long`. Il suffisso `U` può comparire insieme a `L` o `LL`.

Per i ***floating point*** si può scegliere tra notazione decimale e "scientifica":

- `float`: `123.45F`, `1.2345e2F`
- `double`: `123.45`, `1.2345e2`
- `long double`: `123.45L`, `1.2345e2L`
- `void`: <nessuna>
- `std::nullptr_t`: `nullptr`
- Letterali stringa: `"Hello"`

Il tipo associato al letterale è `const char[6]`, cioè un array di 6 caratteri costanti ( $5 + 1$  per il terminatore 0). E' possibile specificare un prefisso di encoding (`u8`, `u`, `U`, `L`) come nel caso delle costanti carattere, che modifica in modo analogo il tipo degli elementi dell'array.

Nel nuovo standard sono stati implementati anche i letterali di stringa grezza (raw string literal), che utilizzano il prefisso `R`, un delimitatore a scelta e le parentesi tonde, come segue: `R"DELIMITATORE(...)" DELIMITATORE"`

Al posto dei ... è possibile inserire qualunque tipo di carattere escludendo le stringhe "DELIMITATORE(" e ")DELIMITATORE", ad esempio posso scrivere: `R"$(Esempio di scrittura all'interno della stringa)$"`

[Torna all'indice](#)

---

## User Defined Literal

Il C++ 2011 ha reso possibile anche la definizione dei cosiddetti letterali definiti dall'utente. Si tratta di una notazione che consente di aggiungere ad un letterale (intero, floating o stringa) un suffisso definito dall'utente: il letterale verrà usato come argomento per invocare una funzione di conversione implicita definita anch'essa dall'utente.

Per esempio, a partire dal C++ 2014, il tipo di dato delle stringhe stile C++ (`std::string`) fornisce la possibilità di usare il suffisso '`s`' per indicare che un letterale stringa deve essere convertito in `std::string`. L'operatore di conversione è definito nel namespace `std::literals`, per cui

```
#include
#include

int main() {
    using namespace std::literals;
    std::cout << "Hello"; // stampa la stringa C (tipo const char[6])
    std::cout << "Hello"s; // stampa la stringa C++ (tipo std::string)
}
```

[Torna all'indice](#)

---

## Gli alias di tipo

Gli alias di tipo, implementati dopo il 2011, utilizzano la keyword `using` e sono utili per riuscire a creare un codice più ordinato, seguendo la filosofia "*write once*".

```
using typeAlias = int;
// in questo caso `typeAlias` sarà un alias per i tipi interi

int main(){
    typeAlias x = 1; // il tipo di `x` sarà quello attribuito a `typeAlias`
}
```

L'utilizzo degli alias può tornare utile in codici lunghi e complessi così da non riscontrare problemi nel caso in cui avvengano dei cambiamenti di tipo.

```
#include

using typeAlias = int;

typeAlias fact(typeAlias n) {
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}
```

```
int main(){
    for(typeAlias i = 0; i < 10; i++){
        std::cout << "fact(" << i << ") = " <
```

Gli alias seguono lo scope del blocco in cui si trovano.

[Torna all'indice](#)

---

## La keyword auto

Nel 2011 è stata attribuita la keyword `auto`, che precedentemente aveva un altro utilizzo, per inizializzazione di una variabile senza esplicitarne il tipo.

```
#include

int main(){
    auto a = 1; // `a` è di tipo intero
}
```

[Torna all'indice](#)

---

## La libreria GMP

Molto spesso i tipi di dato implementati non sono sufficienti per rappresentare le informazioni richieste. Per riuscire a capire qual è il valore massimo che può essere rappresentato si può fare come segue:

```
#include
#include
int main(){

    std::cout << "Il valore massimo rappresentabile da un intero e' " << std::numeric_limits::max() << std::endl;
    std::cout << "Il valore massimo rappresentabile da un long e' " << std::numeric_limits::max() << std::endl;
    std::cout << "Il valore massimo rappresentabile da un long long e' " << std::numeric_limits::max() << std::endl;

    return 0;
}
```

Per "superare" i limiti imposti dai qualificatori di base si possono utilizzare delle librerie apposite. Una di queste è [GNU \(multiple precision library\)](#): una libreria open-source che permette di allocare spazio sulla ram per riuscire a rappresentare i numeri richiesti. Un esempio pratico:

```
#include
#include
#include // "interfaccia" per C++

using typeAlias = mpz_class; // sfrutto un alias

typeAlias fact(typeAlias n){
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}

int main(){

    for(typeAlias i = 0; i < 50; i++)
        std::cout << "fact(" << i << ") = " << fact(i) << std::endl;

    return 0;
}
```

In fase di compilazione devo esplicitare le librerie:

- gmpxx
- gmp (libreria \$C\$)

```
g++ -Wall -Wextra -o fact fact.cpp -lgmpxx -lgmp
```

[Torna all'indice](#)

# Tipi riferimento e tipi puntatore

I tipi riferimento e i tipi puntatore vengono spesso confusi tra loro dai programmati che prendono in considerazione gli aspetti implementativi del linguaggio, in quanto entrambi sono rappresentati internamente utilizzando l'indirizzo dell'oggetto riferito o puntato.

Una visione a più alto livello mette in evidenza alcune differenze, sia a livello sintattico che (cosa più importante) a livello semantico. Un modo intuitivamente semplice (anche se formalmente non esatto) di capirne la differenza è il seguente:

- un **PUNTATORE** è un oggetto, il cui valore (un indirizzo) si può riferire ad un altro oggetto (necessariamente diverso);
- un **RIFERIMENTO** non è un oggetto vero e proprio, ma è una sorta di "nome alternativo" fornito per accedere ad un oggetto esistente.

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. Un riferimento è un'alias di un'altra variabile, ovvero un altro nome per la stessa area di memoria.

In sostanza, la principale differenza tra un puntatore e un riferimento è che un puntatore può essere inizializzato a `nullptr` e può essere ri-assegnato a puntare ad altre aree di memoria, mentre un riferimento deve essere inizializzato al momento della definizione e non può essere ri-assegnato ad un'altra area di memoria. Inoltre, la sintassi per accedere al valore contenuto in un'area di memoria è diversa: per un puntatore si usa l'operatore di dereferenziazione, mentre per un riferimento si usa il riferimento stesso come se fosse la variabile originale.

[Ritorna all'indice](#)

## Osservazioni

Da queste "definizioni" seguono alcune osservazioni:

1. Quando viene **creato un riferimento**, *questo deve sempre essere inizializzato*, in quanto si deve sempre riferire ad un oggetto esistente; in altre parole, non esiste il concetto di "riferimento nullo". In contrasto, un puntatore può essere inizializzato con il letterale `nullptr` (o con il puntatore nullo del tipo corretto, mediante conversioni implicite), nel qual caso NON punterà ad alcun oggetto.
2. Una volta creato un riferimento, **questo si riferirà sempre allo stesso oggetto**; non c'è modo di "riassegnare" un riferimento ad un oggetto differente. In contrasto, durante il suo tempo di vita, un oggetto puntatore (che non sia qualificato `const`) può essere modificato per puntare ad oggetti diversi o a nessun oggetto.
3. Ogni volta che si effettua una **operazione su un riferimento**, in realtà si sta (implicitamente) operando sull'oggetto riferito. In contrasto, nel caso dei puntatori abbiamo a che fare con due oggetti diversi: l'oggetto puntatore e l'oggetto puntato. Eventuali operazioni di lettura e scrittura (comprese le operazioni relative all'aritmetica dei puntatori) applicate direttamente al puntatore accederanno e potenzialmente modificheranno l'oggetto puntato. Per lavorare sull'oggetto puntato, invece, occorrerà usare l'operatore di dereferenziazione, in una delle forme

```
*p      // operator* prefisso  
p->a  // operator-> infisso, equivalente a (*p).a
```

4. Un eventuale qualificatore '`const`' aggiunto al riferimento si applica necessariamente all'oggetto riferito e non al **riferimento stesso**. In contrasto, nel caso del puntatore, avendo due oggetti (puntatore e puntato), è possibile specificare il qualificatore `const` (o meno) per ognuno dei due oggetti. A livello sintattico, nella dichiarazione di un puntatore è possibile scrivere due volte il qualificatore `const`:
  - se `const` sta a sinistra di `*`, si applica all'oggetto puntato;
  - se `const` sta a destra di `*`, si applica all'oggetto puntatore.

[Ritorna all'indice](#)

# Esempi

```
int i = 5;           // oggetto modificabile
const int ci = 5;   // oggetto non modificabile

int& r_i = i;       // ok: posso modificare i usando r_i
const int& cr_i = i; // ok: prometto di non modificare i usando cr_i
int& r_ci = ci;    // errore: non posso usare un riferimento non-const
                   // per accedere ad un oggetto const

const int& cr_ci = ci; // ok: accesso in sola lettura

int& const cr = i; // errore: const può stare solo a sinistra di &

int* p_i;           // p_i e *p_i sono entrambi modificabili
const int* p_ci;    // p_ci è modificabile, *p_ci non lo è
int* const cp_i = &i; // cp_i non è modificabile, *cp_i è modificabile
const int* const cp_ci = &i; // cp_ci e *cp_ci sono entrambi non modificabili

p_i = &i;          // ok
p_i = &ci;         // errore: non posso inizializzare un puntatore a non-const
                   // usando un indirizzo di un oggetto const

p_ci = &i;         // ok: prometto che non modificherò i usando *p_ci
p_ci = &ci;         // ok

cp_i = nullptr;    // errore: cp_i è non modificabile
cp_ci = &ci;        // errore: cp_ci è non modificabile
```

[Ritorna all'indice](#)

---

## Somiglianze

Evidenziate le differenze, possiamo ora discutere alcune somiglianze (magari non tanto banali) tra puntatori e riferimenti.

1. Quando termina il tempo di vita di un puntatore (ad esempio, quando si esce dal blocco di codice nel quale era stato definito come variabile locale), viene distrutto l'oggetto puntatore, ma NON viene distrutto l'oggetto puntato (cosa che potrebbe dare origine ad un memory leak). Analogamente, quando un riferimento va fuori scope, l'oggetto riferito non viene distrutto.

**NOTA - Il caso speciale** Esiste però il caso speciale del riferimento inizializzato con un oggetto temporaneo, che viene distrutto insieme al riferimento stesso (se ne era parlato nella [discussione](#) sul tempo di vita degli oggetti).

2. Analogamente al *dangling pointer* (un puntatore non nullo che contiene l'indirizzo di un oggetto non più esistente), è possibile creare un *dangling reference*, ovvero un riferimento che si riferisce ad un oggetto non più esistente. Il classico esempio è il seguente:

```
struct S { /* ... */ };

S& foo() {
    S s;
    // ...
    return s;
}
```

Si tratta chiaramente di un ***GRAVE ERRORE*** di programmazione: la funzione ritorna per riferimento un oggetto che è stato allocato automaticamente all'interno della funzione stessa; tale oggetto però, viene automaticamente distrutto quando si esce dal blocco nel quale è stato definito e quindi il riferimento restituito al chiamante è invalido. L'approccio corretto è di modificare l'interfaccia della funzione `foo` affinché ritorni per valore, invece che per riferimento.

[Ritorna all'indice](#)

## ODR: One Definition Rule

Quando il codice di un programma deve essere suddiviso in più unità di traduzione, si pone il problema di come fare interagire correttamente le varie porzioni del programma. Intuitivamente, le varie unità di traduzione devono concordare su una interfaccia comune. Tale interfaccia è formata quindi da dichiarazioni di tipi, variabili, funzioni, ecc.

Per ridurre il rischio che una delle unità di traduzione si trovi ad operare con una versione diversa (inconsistente) dell'interfaccia, si cerca di seguire la regola **DRY** ("Don't Repeat Yourself"): le dichiarazioni dell'interfaccia vengono scritte una volta sola, in uno o più header file.

Le varie unità di traduzione includeranno gli header file di cui hanno bisogno, evitando di ripetere le corrispondenti dichiarazioni. Il meccanismo è intuitivamente semplice, ma se usato senza la necessaria cautela, può dare luogo a problemi che, in ultima analisi, si possono ricondurre a violazioni della "One Definition Rule".

La **ODR** (regola della definizione unica) stabilisce quanto segue:

1. Ogni unità di traduzione che forma un programma può contenere non più di una definizione di una data variabile, funzione, classe, enumerazione o template.
2. Ogni programma deve contenere esattamente una definizione di ogni variabile e di ogni funzione non-inline usate nel programma.
3. Ogni funzione inline deve essere definita in ogni unità di traduzione che la utilizza.
4. In un programma vi possono essere più definizioni di una classe, enumerazione, funzione inline, template di classe e template di funzione a condizione che:
  - tali definizioni siano sintatticamente identiche;
  - tali definizioni siano semanticamente identiche.

[Torna all'indice](#)

---

## Esempi

Forniamo alcuni esempi per chiarire i vari punti della ODR; in particolare, ci concentreremo sulle possibili violazioni della regola.

---

### Violazione del punto 1

Definizione multipla di tipo in una unità di traduzione:

```
struct S { int a; };
struct S { char c; double d; };
```

Definizione multipla di variabile in una unità di traduzione:

```
int a;
int a;
```

Il motivo dell'errore è evidente: le due definizioni con lo stesso nome creano una ambiguità. Si noti che si considera il nome completamente qualificato, per cui la seguente **NON è una violazione**, perché si tratta di variabili diverse (`N::a` e `::a`).

```
namespace N { int a; }
int a;
```

Analogamente, per le funzioni è lecito l'overloading, per cui anche la seguente **NON è una violazione**, perché si tratta di funzioni diverse, `int ::incr(int)` e `long ::incr(long)`:

```
int incr(int a) { return a + 1; }
long incr(long a) { return a + 1; }
```

Si noti inoltre che si parla di definizioni. È lecito avere più dichiarazioni della stessa entità, a condizione che solo una di esse sia una definizione (ovvero, le altre devono essere dichiarazioni pure). L'esempio seguente **NON contiene violazioni** ODR:

```
struct S; // dichiarazione pura
struct S { int a; }; // definizione
```

```
struct S;           // dichiarazione pura
S a;               // definizione
extern S a;         // dichiarazione pura
void foo();         // dichiarazione pura
void foo() { }      // definizione

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

[Torna all'indice](#)

---

## Violazione del punto 2

Un caso banale è quello dell'uso di una variabile o funzione che è stata dichiarata ma non è stata mai definita nel programma (zero definizioni): la compilazione in senso stretto andrà a buon fine, ma il linker segnalerà l'errore al momento di generare il codice eseguibile.

Più interessante è il caso delle definizioni multiple (magari pure inconsistenti) effettuate in unità di traduzione diverse:

```
// Siamo in foo.hh
int foo(int a);

// Siamo in file1.cc
#include "foo.hh"
int foo(int a) { return a + 1; }

// Siamo in file2.cc
#include "foo.hh"
int foo(int a) { return a + 2; }

// Siamo in file3.cc
#include "foo.hh"
int bar(int a) { return foo(a); }
```

Il linker, al momento di collegare (l'object file prodotto dalla compilazione di) `file3.cc` con il resto del programma potrebbe indifferentemente invocare la funzione `foo` definita in `file1.cc` o quella definita in `file2.cc`, ottenendo effetti non prevedibili.

Tipicamente, il linker segnalerà l'errore, ma in realtà le regole del linguaggio dicono che non è tenuto a farlo (la formula usata nello standard è "no diagnostic required") e, nel caso, la colpa dell'errore ricade sul programmatore.

[Torna all'indice](#)

---

## Violazione del punto 3

Il punto 3 dice che le funzioni inline devono essere definite ovunque sono usate; il senso della regola è chiaro, se si è compreso il meccanismo dell'[inlining delle funzioni](#), che prevede che la chiamata di funzione possa essere sostituita con l'espansione in linea del corpo della funzione (a scopo di ottimizzazione). Tale espansione è effettuata durante la fase di compilazione in senso stretto, per cui il corpo della funzione deve essere presente in ogni unità di traduzione che contiene una chiamata.

Per quanto detto, il linker deve segnalare come errore il caso di una funzione non-inline che è definita in più unità di traduzione, mentre non deve segnalare errore se la funzione è inline. Come fa a distinguere questi due casi? La risposta la possiamo intuire usando lo strumento `nm` che visualizza il contenuto di un object file.

```
// Siamo in aaa.cc
inline int funzione_inline() { return 42; }
int funzione_non_inline() { return 1 + funzione_inline(); }
```

Compilando con l'opzione `-C` e invocando `nm` sull'object file generato, vediamo quanto segue:

```
$ nm -C aaa.o
0000000000000000 W funzione_inline()
0000000000000000 T funzione_non_inline()
```

La funzione non inline è marcata come simbolo definito (etichetta `T`). La funzione inline, invece, è marcata con l'etichetta `w` (weak symbol). Dal manuale di `nm`:

"W"  
"w" The symbol is a weak symbol that has not been specifically tagged as a weak object symbol.  
When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error.  
When a weak undefined symbol is linked and the symbol is not defined, the value of the symbol is determined in a system-specific manner without error.  
On some systems, uppercase indicates that a default value has been specified.

[Torna all'indice](#)

---

## Violazione del punto 4

Il punto più interessante della ODR è il 4. Si applica a classi, enumerazioni, funzione inline, template di classe e template di funzione, ma per comprendere il problema è sufficiente considerare il caso delle definizioni di classi.

### Esempio di violazione della 4a

```
// Siamo in file1.cc

struct S {
    int a;
    int b;
};      // definizione del tipo S
S s;        // definizione di variabile di tipo S

// Siamo in file2.cc

struct S { int b; int a; }; // definizione del tipo S (sintassi diversa)
extern S s;   // dichiarazione pura della s definita in file1.cc
```

Quando il compilatore lavora su `file2.cc`, viene ingannato dalla diversa definizione del tipo `S`, ma non se ne può accorgere. Non è nemmeno detto che se ne accorga il linker (in ogni caso, non è tenuto a farlo).

Con una piccola variante, è possibile ottenere una violazione della 4b, nella quale i tipi sono sintatticamente identici, ma semanticamente diversi:

### Esempio di violazione della 4b

```
// Siamo in file1.cc

typedef T int;
struct S { T a; T b; }; // definizione del tipo S

// Siamo in file2.cc

typedef T double;
struct S {
    T a;
    T b;
};      // definizione del tipo S
        // (sintassi identica, ma semantica diversa)
```

[Torna all'indice](#)

---

## Il tipo Razionale

Avendo analizzato le clausole che compongono la ODR, rimane da capire cosa va fatto all'atto pratico per garantirne la soddisfazione.

La linea guida principale, già nominata, è la DRY (don't repeat yourself), a volte detta anche "*write once*": scrivere una volta sola le dichiarazioni e/o definizioni, inserendole negli header file, e includere questi dove necessario.

L'esempio seguente mostra che l'approccio non risolve tutti i problemi.

Supponiamo di avere un programma che effettua calcoli matematici. Il programma necessita di utilizzare numeri razionali, per i quali si crea un tipo apposito (`Razionale`), la cui definizione è messa nell'header file `Razionale.hh`:

```
// Siamo in Razionale.hh

class Razionale {
```

```
// ... le cose giuste  
};
```

Una seconda parte del programma deve gestire polinomi a coefficienti razionali, per cui si crea un altro header file:

```
// Siamo in Polinomio.hh  
  
#include "Razionale.hh"  
class Polinomio {  
    // ... le cose giuste, usando anche Razionale  
};
```

In una terza parte del programma, si deve scrivere un algoritmo che effettua calcoli sui polinomi, usando anche i razionali:

```
// Siamo in Calcolo.cc  
  
#include "Razionale.hh"  
#include "Polinomio.hh"  
  
// Codice che usa i tipi Polinomio e Razionale
```

Quando compileremo (l'unità di traduzione corrispondente a) `Calcolo.cc`, otterremo un errore dovuto alla violazione della clausola 1 della ODR. L'unità conterrà infatti *due* definizioni della classe `Razionale`, la prima ottenuta dalla prima direttiva di inclusione e la seconda ottenuta (indirettamente) dalla seconda direttiva di inclusione.

[Torna all'indice](#)

---

## Qual è il modo corretto di risolvere questa situazione?

Un modo sicuramente **sbagliato** (dal punto di vista metodologico) è quello di modificare `Calcolo.cc`, eliminando l'inclusione di `Razionale.hh`:

```
// Siamo in Calcolo.cc  
  
#include "Polinomio.hh"  
  
// Codice che usa i tipi Polinomio e Razionale
```

Questo approccio *sembra* funzionare nella situazione contingente, ma è destinato a creare più problemi di quanti non ne risolva.

In primo luogo, diminuisce la leggibilità del codice, perché non è più così evidente che `Calcolo.cc` dipende anche dall'header file `Razionale.hh` (affidandosi all'inclusione indiretta).

In secondo luogo, occorre considerare cosa succede se il responsabile dello sviluppo della classe `Polinomio` decidesse di modificare l'implementazione della sua classe, per esempio per utilizzare il tipo di dato `Frazione` al posto di `Razionale` (eliminando l'inclusione di questo header file): la compilazione di `Calcolo.cc` fallirebbe anche se nessuno ne ha modificato il file sorgente (cosa che non accadrebbe se venisse mantenuta in `Calcolo.cc` l'inclusione di `Razionale.hh`).

**Occorre quindi consentire ad ogni unità di traduzione di includere tutti gli header file di cui necessita**, trovando una soluzione alternativa al problema delle inclusioni multiple dello stesso header file.

Alcuni compilatori prevedono l'utilizzo di direttive speciali per il processore (che non sono standard), come `#pragma once` che, inserita nell'header file, comunica al preprocessore di evitarne l'inclusione multipla.

Una soluzione che invece è garantita funzionare per qualunque compilatore si basa sull'uso delle **"guardie contro l'inclusione ripetuta"**. In pratica, l'header file `Razionale.hh` viene modificato in questo modo:

```
// Siamo in Razionale.hh  
  
#ifndef RAZIONALE_HH_INCLUDE_GUARD  
#define RAZIONALE_HH_INCLUDE_GUARD 1  
  
class Razionale {  
    // ... le cose giuste  
};  
  
#endif /* RAZIONALE_HH_INCLUDE_GUARD */
```

**La prima volta che il file viene incluso, la "guardia"** (ovvero, il flag del preprocessore `RAZIONALE_HH_INCLUDE_GUARD`) **NON è ancora definita** e quindi il preprocessore procede con l'inclusione. Come prima cosa, viene definita la guardia stessa e poi si includono nell'unità di traduzione tutte le dichiarazioni e definizioni (potenzialmente

elaborando altre direttive di inclusione). Se capitasse, durante il preprocessing di quella stessa unità di traduzione, di ritentare altre volte l'inclusione di `Razionale.hh`, il preprocessore troverebbe la guardia già definita; la condizione di `#ifndef` valuterebbe a falso e quindi non avverrebbe nessuna inclusione ripetuta.

**NOTA:** questa soluzione, per funzionare, deve essere applicata sistematicamente su *tutti* gli header file che fanno parte del programma (`Polinomio.hh`, ecc). Inoltre, occorre prestare attenzione ed evitare di usare la stessa guardia (ovvero, flag del preprocessore con lo stesso nome) per file di inclusione distinti.

**NOTA:** verificare che le guardie contro l'inclusione ripetuta sono utilizzate negli header file della libreria standard distribuiti con `g++`.

Per esempio, nell'header file `iostream`:

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

// ...

#endif /* _GLIBCXX_IOSTREAM */
```

[Torna all'indice](#)

---

## Costrutti del linguaggio

Dopo avere discusso della ODR e delle sue possibili violazioni, può essere utile fare un [elenco dei costrutti del linguaggio che è ragionevole trovare all'interno di un header file](#):

- direttive del preprocessore (inclusione, guardie, macro, ...)
- commenti
- dichiarazioni o definizioni di tipo
- dichiarazioni di variabili
- definizioni di costanti
- dichiarazioni di funzioni
- definizioni di funzioni inline
- dichiarazioni o definizioni di template
- namespace dotati di nome: `namespace N { ... }`
- alias di tipi: `{ typedef std::vector<int> Ints; using Ints = std::vector<int>; }`

[I seguenti costrutti, invece, NON si dovrebbero trovare in un header file:](#)

- definizione di variabili
- definizione di funzioni

### Perchè?

Siccome gli header file sono pensati per essere inclusi in più unità di traduzione, si avrebbe una violazione della ODR (clausola 2).

[Anche i seguenti costrutti NON si dovrebbero trovare in un header file:](#)

- namespace anonimi: `namespace { ... }`
- dichiarazioni di `using`
- direttive di `using`

In questo caso, però, non c'entra la *ODR*. Direttive e dichiarazioni di `using` annullano il meccanismo di protezione dai conflitti di nomi che viene fornito dai `namespace`.

**È lecito farlo** in contesti molto limitati e sotto il nostro controllo (ad esempio, all'interno della definizione di una funzione o all'interno di una unica unità di traduzione). **E considerato cattivo stile** invece farlo in un header file (a meno che non si sia all'interno del corpo di una funzione inline o di un template di funzione), perché il potenziale problema si propagherebbe a tutte le unità di traduzione che includono il nostro header file (direttamente o indirettamente).

[Torna all'indice](#)

# Passaggio argomenti

Nel \$C\$++ 2003, gli argomenti delle funzioni possono essere passati secondo due modalità:

- **Passaggio per valore (T)**: si effettua un copia del valore dell'argomento nel parametro della funzione.
- **Passaggio per riferimento a lvalue** (`const T&` oppure `T&`): il parametro della funzione è un riferimento che viene inizializzato con l'argomento stesso, senza effettuare una copia.

La linea guida, semplice, prevede di passare per copia solo gli oggetti piccoli, per i quali la copia stessa non è costosa, mentre gli oggetti potenzialmente grandi sono passati per riferimento a costante. Nel caso sia necessario modificare direttamente l'argomento in input, si opta per il passaggio a riferimento modificabile.

Per il **valore di ritorno**, generalmente si opta per il ritorno per valore, perché non si possono restituire riferimenti a variabili che sono allocate automaticamente dalla funzione (il chiamante otterrebbe dei riferimenti *dangling*). Un caso in cui si possono restituire riferimenti (e quindi evitare copie costose) è quello in cui siamo sicuri che il riferimento è ad un oggetto il cui tempo di vita continuerà sicuramente anche dopo la chiamata di funzione.

## Ivalue vs rvalue

In \$C\$++, un **lvalue** è un'espressione che rappresenta un oggetto identificabile nella memoria del computer, cioè un valore che ha un indirizzo di memoria. Ad esempio, una variabile, un elemento di un array o una funzione che restituisce un riferimento a un oggetto sono tutti esempi di *lvalue*.

D'altra parte, un **rvalue** è un'espressione che rappresenta un valore che non ha un indirizzo di memoria associato. Ad esempio, un valore costante come 5, una stringa letterale come `hello` o il risultato di una chiamata a una funzione che restituisce un valore sono tutti esempi di *rvalue*.

In breve, **la differenza principale tra lvalue e rvalue sta nell'indirizzo di memoria**. Gli *lvalue* sono identificabili dalla memoria e possono essere modificati, mentre gli *rvalue* sono solo valori temporanei che non possono essere modificati. Inoltre, alcune operazioni in \$C\$++ richiedono un *lvalue*, come l'assegnazione di un valore a una variabile, mentre altre richiedono un *rvalue*, come l'uso di un valore costante in un'operazione aritmetica.

Approfondimento: [[11-lvalue\_rvalue]]

**NOTA:** a partire dal \$C\$++ 2011 sono stati introdotti i riferimenti a *rvalue* e, di conseguenza, la possibilità di passare un argomento per riferimento a *rvalue*. L'argomento (un po' tecnico) verrà affrontato quando parleremo della gestione delle risorse.

**NOTA:** a volte, in maniera impropria, si parla di passaggio di un argomento "*per puntatore*". Tecnicamente, si tratta di un caso specifico del passaggio per valore (il valore del puntatore, ovvero un indirizzo). Questo modo di procedere è un'eredità del linguaggio \$C\$ ed è spesso rimpiazzabile dal passaggio per riferimento. Un argomento di tipo puntatore continua ad avere senso quando l'argomento è opzionale: in questo caso, passando il puntatore nullo si segnala alla funzione che quell'argomento non è di interesse per una determinata chiamata.

[Torna all'indice](#)

# Array e puntatori

---

## Array

Quando si usa una espressione di tipo array di  $T$ , viene applicato il *type decay* e si ottiene il puntatore al primo elemento dell'array. Questa conversione (trasformazione di lvalue) è necessaria per evitare l'uso di copie costose quando un array viene passato come argomento ad una funzione: si passa (per valore) il puntatore.

Il legame tra array e puntatori è molto forte: basti considerare che la sintassi dell'indicizzazione di un array non è altro che una abbreviazione per un utilizzo semplificato dell'aritmetica dei puntatori.

---

## Esempio

Si considerino le due variabili:

```
int a[100];
int b = 5;
```

L'espressione  $a[b]$  è in tutto e per tutto equivalente all'espressione  $*(a + b)$ . In questa seconda espressione si usa infatti l'array  $a$ , che decade al puntatore al suo primo elemento; poi si somma  $b$  al puntatore, che corrisponde a muoversi in avanti nell'array di 5 posizioni, ottenendo l'indirizzo del sesto elemento. Infine si dereferenzia l'indirizzo, ottenendo il sesto elemento dell'array.

Siccome la somma è commutativa, lo stesso risultato lo si ottiene anche usando l'espressione  $*(b + a)$  che per quanto detto sopra risulta essere equivalente a  $b[a]$ .

**NOTA BENE:** chiaramente, un programmatore sensato dovrebbe astenersi dall'utilizzare costrutti che hanno il solo scopo di sorprendere (o trarre in inganno).

**NOTA:** questo "trucco" funziona solo con gli array; non funziona con altri contenitori, quali `std::vector<T>`.

[Torna all'indice](#)

---

## Aritmetica dei puntatori

Se  $\text{ptr}$  è un puntatore (del tipo corretto) che indirizza un elemento all'interno di un array, allora le espressioni

```
ptr + n    // muoviti in avanti di n posizioni
n + ptr    // idem
ptr - n    // muoviti all'indietro di n posizioni
```

sono legittime nella misura in cui il puntatore risultante dopo il movimento continua a puntare ad un elemento dell'array (cioè non si è andati oltre il limite iniziale o finale) oppure punti all'indirizzo immediatamente successivo alla fine dell'array.

Dati due puntatori  $\text{ptr1}$  e  $\text{ptr2}$  (del tipo corretto) che indirizzano elementi dello stesso array, l'espressione  $\text{ptr1} - \text{ptr2}$  indica la distanza tra i due puntatori, ovvero il numero di elementi che li separa (si **noti** che la distanza potrebbe essere negativa).

L'aritmetica dei puntatori si presta alla definizione di un importante idioma di programmazione relativo all'iterazione su array.

[Torna all'indice](#)

## Esempio

```
int a[100];

// iterazione basata su indice
for (int i = 0; i != 100; ++i) {
    // fai qualcosa con a[i]
}

// iterazione basata su puntatore
for (int* p = a; p != a + 100; ++p) {
    // fai qualcosa con *p
    // I puntatori sono iteratori per gli array
}
```

La seconda forma si presta bene a generalizzazioni che non richiedono di conoscere il punto di inizio dell'array e la sua dimensione. Se sono sicuro che `p1` e `p2` sono puntatori validi sull'interno dell'array e sono anche sicuro che `p1` non viene dopo `p2`, allora posso iterare su tutti gli elementi compresi tra l'elemento puntato da `p1` (incluso) e l'elemento puntato da `p2` (escluso), nel modo seguente:

```
// iterazione basata su coppie di puntatori
for ( ; p1 != p2; ++p1) {
    // fai qualcosa con *p1
}
```

Spesso, per la coppia `p1` e `p2` si usano i nomi `first` e `last`, con l'accortezza di ricordarsi che `last`, in effetti, si riferisce alla posizione *successiva* all'ultimo elemento che si vuole processare.

Se si vuole specificare una sequenza vuota, è sufficiente fornire una coppia di puntatori identici (ottenendo quindi un ciclo che non effettua alcuna iterazione). Questo i dia ma è stato esteso nel \$C\$++ al caso degli iteratori sulle sequenze generiche e sui contenitori della libreria standard (e quindi è di estrema rilevanza per la programmazione in \$C\$++)�

[Torna all'indice](#)

# Tipi dati concreti

---

## Che cos'è?

In \$C\$++, un **tipo di dato concreto** è un tipo di dato che rappresenta un valore effettivo memorizzato in un'area di memoria specifica. Ciò significa che un tipo di dato concreto ha un'allocazione di memoria definita e fissa e che il valore può essere modificato tramite un puntatore o un riferimento a tale area di memoria. [[14.2-esercizio\_lifetime]] Ad esempio, un tipo di dato concreto in \$C\$++ potrebbe essere un `int`, un `float` o un `char`. Questi tipi di dati hanno un'allocazione di memoria fissa e possono essere modificati direttamente utilizzando un puntatore o un riferimento.

D'altra parte, un **tipo di dato astratto** in \$C\$++ rappresenta un concetto astratto, come un oggetto o una collezione di oggetti, e non ha un'allocazione di memoria fissa. Ad esempio, una classe definita dall'utente è un tipo di dato astratto poiché rappresenta un'entità concettuale e non ha un'allocazione di memoria fissa. Tuttavia, un oggetto di una classe ha un'allocazione di memoria fissa e può essere considerato come un tipo di dato concreto.

In sintesi, i tipi di dato concreti in \$C\$++ rappresentano valori effettivi che sono memorizzati in una specifica area di memoria e possono essere modificati direttamente.

[Torna all'indice](#)

---

## La classe Razionale

Un esempio di dato concreto è la classe Razionale. Prima della definizione dei metodi bisogna stabilire quali funzioni si vogliono rendere disponibili all'utente: per fare ciò si sfruttano dei *"test"* che fungeranno da linee guida per la scrittura dei metodi.

Per compilare:

```
g++ -std=c++11 -Wall -Wextra -o testRazionale.o -c testRazionale.cc
```

In seguito alla compilazione si andranno a riscontrare degli errori che fungeranno da *"todo list"* per quello che c'è da fare.

Un esempio di primo errore è l'assenza del file `Razionale.hh` in `Razionale.cc`.

Con `-std=c++11` si intende lo standard \$C\$++ del 2011.

[Torna all'indice](#)

---

## Razionale.hh

Definiamo l'header file, nel quale sarà presente il namespace Numerica contenente la classe Razionale:

```
#ifndef NUMERICA_RAZIONALE_HH_INCLUDE_GUARD
#define NUMERICA_RAZIONALE_HH_INCLUDE_GUARD 1

namespace Numerica {
    class Razionale {
        Razionale() = default;
        Razionale(const Razionale&) = default;
        Razionale(Razionale&&) = default;
        Razionale& operator=(const Razionale&) = default;
        Razionale& operator=(Razionale&&) = default;
        ~Razionale() = default;

        using Intero = long;
        Razionale(Intero n, Intero d = 1);
    }; // end class Razionale
} // end namespace Numerica

#endif NUMERICA_RAZIONALE_HH_INCLUDE_GUARD
```

[Torna all'indice](#)

---

## testRazionale.cc

Definiamo ora invece il file .cc contenente il test. Come prima cosa dobbiamo includere l'header file:

```
#include "Razionale.hh"
```

Successivamente creiamo una funzione `test01` di tipo `void`, in cui andremo a fare i nostri test. Ora importiamo il namespace Numerica:

```
using Numerica::Razionale;
```

Ora inseriamo i vari tipi di costruttori:

```
Razionale r; // costruttore default
Razionale r1(r); // costruttore di copia
Razionale r2 = r; // costruttore di assegnamento
Razionale r3 { r }; // costruttore di copia (C++11)
Razionale r4 { r }; // costruttore di copia (C++11)
Razionale r5(1, 2); // costruzione diretta
Razionale r6 {1, 2}; // costruzione diretta (C++11)
Razionale r7(1); // costruzione diretta
Razionale r8{1}; // costruzione diretta (C++11)
Razionale r9 = 1234; // costruzione implicita (da evitare!), fa una conversione da int a long e da long a Razionale
Razionale r10 = true; // Fa la stessa cosa. Bisogna usare la parola chiave 'explicit' davanti al costruttore
```

I tipi di assegnamento:

```
r = r1; // assegnamento di copia
r = Razionale(1); // assegnamento per spostamento
r2 = r1 = r; // concatenazione assegnamenti (da evitare), r2 = (r1 = r) -> r2 = r1
```

Gli operatori aritmetici:

```
// Binari
r1 = r + r;
r1 = r - r;
r1 = r * r;
r1 = r / r;

// Unari
r1 = -r;
r1 = +r;
```

Gli operatori di assegnamento:

```
r += r;
r -= r;
r *= r;
r /= r;

// Incremento e decremento (pre o post)
++r;
r1 += ++r;
r++;
--r;
r--;
```

Gli operatori relazionari:

```
bool b;
b = (r == r);
b = (r != r);
b = (r < r);
b = (r <= r);
b = (r >= r);
```

```
b = (r > r);
```

E infine le operazioni di input e di output:

```
std::cin >> r;
std::cin >> r >> r1;
std::cout << r << std::endl;
```

Maggiori informazioni nel file `./code/testRazionale.cc`

[Torna all'indice](#)

---

## Problema: le conversioni implicite

Le conversioni implicite in C++ si verificano quando il compilatore converte automaticamente un valore di un tipo di dato in un altro tipo di dato senza che sia necessario esplicitare una conversione tramite un cast.

Un esempio:

```
void foo(int a, bool b);
void bar(){
    foo(3.7, true);
    foo(true, 3.7);
}
```

In questo caso, alla prima chiamata della funzione `foo(3.7, true)` si presenta una conversione implicita di un `float` (3.7) ad un intero. Nella seconda chiamata `foo(true, 3.7)` si presenta una conversione implicita di un booleano ad un intero, e di un `float` ad un booleano.

È importante notare che le conversioni implicite possono portare a comportamenti indesiderati o a errori di runtime se non si tiene conto delle regole di conversione. Pertanto, è consigliabile evitare le conversioni implicite quando possibile e utilizzare invece le conversioni esplicite tramite cast quando necessario.

Vediamo un esempio della classe Razionale:

```
void foo(const Razionale& r);
void bar(){
    foo(true);
}
```

Questo non rispetta il [principio di minima sorpresa](#).

Le linea guida della progettazione semplice è il dover risolvere un problema alla volta. Se bisogna risolvere più problemi, prima li risolvo indipendentemente e poi li unisco.

[Torna all'indice](#)

# Esempio esercizio esame (appello 20050222):

La classe templatica Set è intesa rappresentare un insieme di elementi di tipo T. L'implementazione della classe si basa sulla manipolazione di liste ordinate (senza duplicati). L'interfaccia della classe presenta numerosi problemi.

Cercare di individuarne il maggior numero ed indicare come possono essere risolti (riscrivendo l'interfaccia).

```
template
class Set : public std::list {
public:
    // Costruisce l'insieme vuoto.
    Set();
    // Costruisce il singoletto che contiene t.
    Set(T t);
    Set(Set y);
    void operator=(Set y);
    virtual ~Set();

    unsigned int size();
    bool is_empty();
    bool contains(Set y);

    T& min();
    void pop_min();
    void insert(T z);
    void union_assign(Set y);
    void intersection_assign(Set y);

    void swap(Set y);
    std::ostream operator<<(std::ostream os);

private:
// ...
};
```

Risoluzione:

```
template
class Set {
private:
    std::list m_list; // m: sta per dato membro
    // In questo modo garantisco l'invariante di classe
public:
    // Costruisce l'insieme vuoto.
    Set();

    // Costruisce il singoletto che contiene t.
    // explicit è per evitare la conversione implicita
    explicit Set(const T & t);

    // Se lo avessi lasciato senza il const sarebbe andato in un loop
    // chiamando se stesso all'infinito
    Set(const Set & y);

    // se non voglio permettere la concatenazione di assegnamenti
    // il tipo di ritorno della funzione deve essere void, in caso contrario
    // il tipo deve essere Set& (commento)
    void operator=(const Set & y);
    // Set& operator=(const Set & y);

    // Distruttore
    virtual ~Set();

    // I tre metodi seguenti non vanno a modificare i dati privati, quindi
    // si segue la "regola" del const correctness

    unsigned int size() const;
    // I due metodi successivi sono definiti predicati
    bool is_empty() const;
    bool contains(const Set & y) const;

    // Generalmente il metodo che segue viene reso disponibile in
    // sola lettura e, se non si rischia di violare l'invariante di classe,
    // in lettura e in scrittura
```

```
const T& min() const; // L'insieme restituito non è modificabile
// T& min(); // L'insieme restituito è modificabile, quindi
// potrebbe rendere compromissibile l'invariante di classe

void pop_min();
void insert(const T& z);
void union_assign(const Set& y);
void intersection_assign(const Set& y);

void swap(Set&& y);

// Se il metodo seguente fosse implementato all'interno della classe,
// comporterebbe una chiamata del tipo s << std::cout;
// così violando il principio di sorpresa minima
// std::ostream& operator<<(std::ostream& os) const;
};

template
std::ostream& operator<<(std::ostream& os, const Set& t);
```

# **lvalue e rvalue**

---

## **Categorie di espressioni: lvalue e rvalue**

Le espressioni del \$C\$++ possono essere classificate in a) **lvalue** (*left value*) b) **xvalue** (*expiring lvalue*) c) **prvalue** (*primitive rvalue*)

L'unione di lvalue e xvalue forma i **glvalue** (generalized lvalue). L'unione di xvalue e prvalue forma gli **rvalue** (right value).

Intuitivamente, un glvalue è una espressione che permette di stabilire l'identità di un oggetto in memoria. Il nome "left value", in origine, indicava che tali espressioni potevano comparire a sinistra dell'operatore di assegnamento.

Esempio:

```
int i;
int ai[10];
i = 7; // l'espressione i è un lvalue (e quindi un glvalue)
ai[5] = 7; // l'espressione ai[5] è un lvalue (e quindi un glvalue)
```

![[lvalue\_rvalue.png]]

*m*: movable *i*: has identity

[Torna all'indice](#)

---

## **xvalue**

Un **xvalue** è un glvalue che denota un oggetto le cui risorse possono essere riutilizzate, tipicamente perché sta terminando il suo lifetime (expiring lvalue). Un lvalue è un glvalue che non sia un xvalue.

Esempio:

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

*m* verrà distrutto automaticamente in uscita dal blocco nel quale è stato creato; il valore ritornato dalla funzione non è *m*, ma una sua "copia".

```
void foo2() {
    Matrix m1;
    m1 = foo1(); // l'espressione foo1(), cioè il risultato ottenuto
                  // dalla chiamata di funzione, è un xvalue
}
```

La soluzione del \$C\$++11 prevede non di copiare gli xvalue, ma di spostarli in una nuova locazione, dato che questi non sono più richiesti dalla funzione chiamata. In particolare, non adottando questo approccio, le copie sarebbero due:

- *m* viene copiato nella locazione di ritorno della funzione `foo1()`
- l'oggetto temporaneo restituito da `foo1()` viene copiato in *m1*

[Torna all'indice](#)

---

## **prvalue**

Un prvalue è una espressione che denota un valore "primitivo", ovvero un valore costante o il risultato di una computazione. Il nome "right value", in origine, indicava che tali espressioni potevano comparire *solo* a destra dell'operatore di assegnamento (ovvero, espressioni che darebbero errore se comparissero a sinistra). Intuitivamente, un prvalue *NON* identifica un oggetto in memoria e quindi non è lecito assegnarvi un valore e non ha nemmeno senso prenderne l'indirizzo.

Esempio:

```
int i;
i = 5;      // l'espressione 5 è un prvalue (e quindi un rvalue)
i = 4 + 1; // l'espressione 4 + 1 è un prvalue (e quindi un rvalue)
i = i + 1; // l'espressione i + 1 è un prvalue (e quindi un rvalue)
```

In alcuni casi, un prvalue può essere "materializzato", creando un oggetto temporaneo (un lvalue) che viene inizializzato con il valore del prvalue. Questo è quello che succede, per esempio, quando ad una funzione che ha un argomento di tipo riferimento a costante viene passato un prvalue.

Esempio:

```
void foo(const double& d);
void bar() {
    foo(0.5);
}
```

Qui sopra 0.5 è un rvalue; viene materializzato in un oggetto temporaneo (un lvalue) con cui viene inizializzato il riferimento a lvalue d.

La classificazione delle espressioni in lvalue, xvalue e prvalue è rilevante per capire la differenza tra riferimenti a lvalue (`T&`) e riferimenti a rvalue (`T&&`). Questi ultimi sono stati introdotti nel \$C\$++11 per risolvere problemi tecnici del linguaggio che impedivano di fornire implementazioni efficienti per alcuni costrutti.

[Torna all'indice](#)

---

## Ivalue vs rvalue

Nel \$C\$++ 2003, ogni classe era fornita (se non veniva fatto qualcosa per disabilitarle) di 4 funzioni speciali:

- costruttore di default
- costruttore di copia
- assegnamento per copia
- distruttore

```
struct Matrix {
    Matrix(); // costruttore di default
    ~Matrix(); // distruttore

    Matrix(const Matrix&); // costr. di copia (copy ctor)
    Matrix& operator=(const Matrix&); // assegn. per copia (copy assign.)

    // ... altro
};
```

Una funzione che avesse voluto prendere in input un oggetto `Matrix` e produrre in output una sua variante modificata (senza modificare l'oggetto fornito in input), doveva tipicamente ricevere l'argomento per riferimento a costante e produrre il risultato per valore:

```
Matrix bar(const Matrix& arg) {
    Matrix res = arg; // copia (1)
    // modifica di res
    return res; // ritorna una copia (2)
}
```

Questo era fonte di inefficienze, perché:

1. Non c'era un modo semplice per il chiamante di comunicare il fatto che, in alcuni casi (non tutti), la risorsa passata in input non era più di suo interesse e quindi poteva essere modificata in loco, invece di effettuare la prima copia.
2. Non c'era modo semplice per la funzione per ritornare l'oggetto `res` senza fare la seconda copia (si noti che non è possibile ritornare per riferimento, perché si creerebbe un dangling reference).

Nel C++11, alle 4 funzioni speciali delle classi ne sono state aggiunti altre due:

- costruttore per spostamento (move constructor), e
- assegnamento per spostamento (move assignment)

che lavorano su riferimenti a rvalue:

```
struct Matrix {
    Matrix(); // costruttore di default
    ~Matrix(); // distruttore

    Matrix(const Matrix&); // costr. di copia (copy ctor)
    Matrix& operator=(const Matrix&); // assegn. per copia (copy assign.)

    Matrix(Matrix&&); // costr. per spostamento (move ctor)
    Matrix& operator=(Matrix&&); // assegnam. per spostamento (move assign.)

    // ... altro
};
```

Intuitivamente, una funzione che riceve come argomento un riferimento a rvalue (`Matrix&&`) sa che l'oggetto riferito può essere solo un prvalue o un xvalue. In entrambi i casi, le risorse contenute nell'oggetto non possono essere utilizzate da altri e quindi possono essere spostate dall'oggetto (si potrebbe dire "rubate" all'oggetto, che ne era il proprietario) invece che copiate, guadagnando in efficienza.

Riconsideriamo l'esempio precedente, assumendo che sia disponibile il costruttore per spostamento per `Matrix`:

```
Matrix bar(const Matrix& arg) {
    Matrix res = arg; // copia (1)
    // modifica di res
    return res; // sposta (non copia)
}
```

Il compilatore si accorge che, nella `return res`, l'espressione `res` è un xvalue e quindi utilizza il costruttore di spostamento (invece della copia) per restituirlo al chiamante.

Volendo, è possibile ottimizzare anche la prima copia, fornendo una versione alternativa (in overloading) della funzione `bar`:

```
Matrix bar(Matrix&& arg) {
    // modifica in loco di arg
    return arg; // sposta (non copia)
}
```

Questa seconda versione verrà invocata quando alla funzione viene passato un rvalue (che potrà essere modificato direttamente), mentre la prima versione verrà usata per gli lvalue.

E' anche possibile fondere le due versioni in una sola, usando il passaggio dell'argomento per valore:

```
Matrix bar(Matrix arg) {
    // modifica in loco di arg
    return arg; // sposta (non copia)
}
```

In questo terzo caso, all'atto di effettuare il passaggio dell'argomento alla funzione `bar`, vi sono due possibilità:

1. il chiamante fornisce un lvalue: verrà utilizzato il costruttore di copia sull'argomento, comportandosi come nel caso di `Matrix bar(const Matrix& arg)`;
2. il chiamante fornisce un rvalue: verrà utilizzato il costruttore per spostamento, senza copie, come nel caso di `Matrix bar(Matrix&& arg)`.

## La funzione `std::move`

Supponiamo che il chiamante si trovi a dovere invocare la funzione `bar` discussa sopra con un lvalue `m` di tipo `Matrix`, ma non è interessato a preservare il valore di `m` e quindi lo vorrebbe "spostare" nella funzione `bar`, evitando la copia (costosa e inutile).

Se si usa la chiamata:

```
bar(m);
```

siccome `m` è un lvalue verrebbe comunque invocato (almeno una volta) il costruttore per copia. Per evitarla, occorre un modo per convertire il tipo di `m` da riferimento a lvalue (`Matrix&`) a riferimento a rvalue (`Matrix&&`): questo è esattamente l'effetto ottenuto usando la funzione di libreria `std::move`.

```
bar(std::move(m));
```

Si noti che la `std::move(m)` NON "muove" nulla: piuttosto, trasformando un lvalue in rvalue, lo rende "movable" (spostabile); lo spostamento vero e proprio viene effettuato durante il passaggio del parametro.

# Overloading

---

## Definizione e meccanismo di risoluzione

Il \$C\$++ supporta il concetto di **overloading** (sovraffunzione) per il nome delle funzioni: è possibile definire più funzioni che condividono lo stesso nome e si differenziano solo per il numero e/o il tipo degli argomenti.

L'overloading è utile, in generale, per evitare di dovere fornire un nome distinto a funzioni che, in buona sostanza, fanno operazioni del tutto analoghe, solo su argomenti di tipo diverso.

Per esempio, nella libreria matematica del \$C\$ (linguaggio di programmazione che *NON* supporta il concetto di overloading), abbiamo le tre funzioni

```
float sqrtf(float arg);
double sqrt(double arg);
long double sqrtl(long double arg);
```

che calcolano la radice quadrata di un float, un double e un long double. La differenziazione nel nome risulta alquanto artificiosa. In contrasto, nella libreria standard del \$C\$++ (header file `cmath`) si trovano le tre funzioni:

```
float sqrt(float arg);
double sqrt(double arg);
long double sqrt(long double arg);
```

Per il programmatore è più semplice ricordare un solo nome. La differenza tra i due approcci può diventare ancora più evidente se si pensa ad una funzione di stampa (per esempio `print`) che debba essere applicata a dozzine o centinaia di tipi di dato diversi.

Va comunque notato che anche il \$C\$ supporta una forma ristretta di overloading, che però è confinata al caso degli operatori definiti sui tipi built-in. Per esempio, nell'espressione `v + 1` l'operatore `+` corrisponde alla somma di interi se `v` è di tipo intero, alla somma di double se `v` è di tipo double, ecc...

Oltre ai suddetti motivi di comodità, vedremo come l'overloading di funzione risulterà essenziale quando dovremo scrivere codice generico (utilizzando template di classe e template di funzione).

Quando però si passa dal ruolo di utente di una interfaccia software al ruolo di sviluppatore dell'interfaccia stessa, occorre prestare molta attenzione a non creare un insieme di funzioni in overloading che sia "fuorviante" per l'utente dell'interfaccia.

È quindi necessario capire a fondo i meccanismi della cosiddetta "**risoluzione dell'overloading**", che è il processo seguito dal compilatore all'atto di esaminare ogni singola chiamata di funzione allo scopo di stabilire quale delle funzioni dichiarate debba essere invocata per quella chiamata.

[Torna all'indice](#)

---

## Le tre fasi della risoluzione dell'overloading di funzione

Come detto, la **risoluzione dell'overloading** è un processo svolto staticamente dal compilatore, quando trasforma in assembler il codice contenuto nell'unità di traduzione (prodotta dal preprocessore). In particolare, non può essere influenzata dalle informazioni presenti in altre unità di traduzione (compilazione separata) e neanche dalle informazioni disponibili a tempo di esecuzione.

Daremo una descrizione abbastanza dettagliata (ma comunque incompleta) del processo, che si suddivide in tre fasi. Queste fasi vengono ripetute per ogni singola chiamata di funzione presente nel codice sorgente:

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

## Fase 1: le funzioni candidate

L'insieme delle **funzioni candidate** per una specifica chiamata di funzione è un sottoinsieme delle funzioni che sono state dichiarate all'interno dell'unità di traduzione. In particolare, le funzioni candidate:

1. hanno lo stesso nome della funzione chiamata (non importa numero e tipo degli argomenti)
2. sono visibili nel punto della chiamata

Nella valutazione del primo punto (*nome*) occorre comunque tenere presente che, per gli operatori, la sintassi della chiamata di funzione può variare nella forma:

- operatore prefisso: `++a`
- operatore postfisso: `a[5]`
- operatore infisso: `a - b`
- sintassi funzionale: `operator*(a, b)`

mentre la corrispondente dichiarazione utilizzerà sempre la forma del nome esteso (rispettivamente, `operator++`, `operator[]`, `operator-` e `operator*`).

Più interessante (e complicato) è il secondo punto, riguardante la visibilità delle dichiarazioni nel punto di chiamata. Occorre infatti prestare attenzione alle seguenti casistiche:

1. **Nel caso di invocazione di un metodo di una classe mediante la sintassi `obj.metodo`** (oppure `ptr->metodo`), se l'oggetto `obj` (oppure il puntatore `ptr`) hanno come tipo statico `S`, allora la ricerca inizierà nello scope di classe `S`.

Si noti che viene considerato il tipo *statico* (cioè il tipo noto a tempo di compilazione), mentre viene ignorato il tipo *dinamico* (che è noto solo a tempo di esecuzione).

Esempio:

```
struct S { /* ... */ };
struct T : public S { /* ... */ };
S* ptr = new T;
ptr->foo(); // chiamata
// ptr ha tipo statico S* e tipo dinamico T* (esempio Java Quadr./Rett.)
// quindi la ricerca di foo avviene a partire dallo scope di S
// (eventuali funzioni T::foo non sono visibili nel punto di chiamata)
```

2. **Nel caso di qualificazione della funzione chiamata**, la ricerca delle funzioni candidate inizierà nel corrispondente scope. Esempio:

```
namespace N {
    void foo(int);
}
void foo(char); // dichiarazione NON visibile per la chiamata
int main() {
    N::foo('c'); // la ricerca inizia nello scope di namespace N
}
```

3. **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::foo
// non va in overloading, perché viene nascosta (hiding);
```

4. **Attenzione ad eventuali dichiarazioni e direttive di using**, che modificano la visibilità delle dichiarazioni: nell'esempio precedente, se nella classe T fosse aggiunto `using S::foo;` allora `S::foo` e `T::foo` sarebbero entrambe visibili e andrebbero in overloading.

## 5. Attenzione all'ADL (Argument Dependent Lookup)

[Torna all'indice](#)

---

### ADL - Argument Dependent Lookup

La regola **ADL** (Argument Dependent Lookup), detta anche *Koenig's lookup*, stabilisce che

1. nel caso di una chiamata di funzione NON qualificata,
2. se vi sono (uno o più) argomenti "arg" aventi un tipo definito dall'utente (cioè hanno tipo `struct/class/enum S`, o riferimento a `S` o puntatore a `S`, possibilmente qualificati) e
3. il tipo suddetto è 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  
}
```

1. 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).
2. 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 BENE:** 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).

[Torna all'indice](#)

---

### Fase 2: selezione 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 è utilizzabile, **è necessario verificare che:**

1. il numero degli argomenti (nella chiamata di funzione) sia compatibile con il numero parametri (nella dichiarazione di funzione);
2. ogni argomento (nella chiamata di funzione) abbia un tipo compatibile con il corrispondente parametro (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.

Con riferimento alla compatibilità dei tipi argomento/parametro, occorre tenere conto che, oltre al caso della corrispondenza perfetta tra i due tipi (chiamata conversione identità o match perfetto, anche se tecnicamente in questo caso non si effettua nessuna conversione), è applicabile tutta una serie di conversioni implicite che potrebbero consentire di convertire il tipo dell'argomento nella chiamata nel tipo del parametro nella dichiarazione di funzione.

Si coglie l'occasione per ricordare la classificazione delle conversioni implicite in:

1. corrispondenze esatte (identità, trasformazioni di lvalue, qualificazioni);
2. promozioni;
3. conversioni standard;
4. conversioni definite dall'utente.

[Torna all'indice](#)

---

## Sequenza di conversioni

Quando si verifica se una funzione è utilizzabile, per ogni argomento della chiamata è possibile effettuare una sequenza di conversioni (i.e., non ci si limita ad usare una sola conversione隐式).

Una sequenza di conversione "standard" è composta da: `$$ \text{trasf. di lvalue} + (\text{promozione o conv. standard}) + \text{qualificazione} $$` Le varie componenti sono opzionali, ma se presenti devono essere nell'ordine specificato.

Esempio: supponendo che la classe D è derivata dalla classe base B,

```
double d = 3.1415; // un lvalue di tipo double
void foo(int);      // funzione che accetta un intero per valore

foo(d); // chiamata
```

la funzione `void foo(int)`, se candidata, è anche utilizzabile.

Si applica prima una trasformazione di lvalue (nello specifico, una trasformazione da lvalue a rvalue, che corrisponde alla lettura del valore contenuto nella locazione d) e quindi una conversione standard (da double a int).

Una sequenza di conversione "utente" è composta da: `$$ \text{seq. conv. standard} + \text{conv. utente} + \text{seq. conv. standard} $$`

[Torna all'indice](#)

---

## Fase 3: selezione della migliore utilizzabile

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 un'unica vincitrice, quella funzione è la migliore utilizzabile; altrimenti si ottiene un errore di compilazione (*chiamata ambigua*).

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 argomenti. X è preferibile a Y se:

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

Fissato un argomento `A_i`, la funzione X vince rispetto alla funzione Y se la sequenza di conversioni `X_i` usata da X per `A_i` vince rispetto alla sequenza `Y_i` usata da Y.

Le regole del linguaggio per stabilire se una sequenza `x_i` vince sulla sequenza `y_i` sono abbastanza intricate e non uniformi (sono elencate un certo numero di eccezioni). Nello standard C++14 la loro spiegazione occupa almeno 3 pagine dense. A scopo didattico, usiamo quindi una versione semplificata.

Una sequenza `x_i` vince sulla sequenza `y_i` se la peggiore conversione `x_worst` usata in `x_i` vince sulla peggiore conversione `y_worst` usata in `y_i`. Una conversione `x_worst` vince sulla conversione `y_worst` se ha un "rank" migliore (corrispondenze esatte vincono sulle promozioni, che vincono sulle conversioni standard, che vincono sulle conversioni utente).

[Torna all'indice](#)

---

## Esempio

![[overloading\_esempio.jpg]]

[Torna all'indice](#)

---

## Ulteriori osservazioni

1. Le regole fin qui esposte NON prendono in considerazione alcuni casi speciali che si applicano in presenza di funzioni candidate ottenute istanziando (o specializzando) template di funzione. Questi casi verranno introdotti al momento opportuno.
2. Vale la pena sottolineare che, quando si scelgono le funzioni candidate per una chiamata, il numero e il tipo dei parametri della funzione NON sono considerati in alcun modo (entrano in gioco solo nella seconda fase, quando si restringe l'insieme delle candidate al sottoinsieme delle funzioni utilizzabili).
3. Analogamente, è opportuno sottolineare che, nel caso di invocazione di un metodo di una classe, il fatto che tale metodo sia dichiarato con accesso `public`, `private` o `protected` NON ha nessun impatto sul processo di risoluzione dell'overloading. Il processo determina la migliore funzione utilizzabile (se esiste); in seguito, se la funzione scelta non è accessibile per il chiamante, verrà segnalato un errore di compilazione, che però NON ha nulla a che fare con la risoluzione dell'overloading (in particolare, la migliore funzione utilizzabile ma non accessibile NON viene mai sostituita da un'altra funzione utilizzabile e accessibile).

[Torna all'indice](#)

# Conversioni implicite di tipo

Le conversioni implicite del \$C\$++ si possono suddividere in 4 categorie:

1. Le corrispondenze "esatte"
2. Promozioni
3. Conversioni standard
4. Conversioni implicite definite dall'utente

Le categorie sarebbero 5, ma per semplicità *NON* prendiamo in considerazione le conversioni che si applicano nel caso della sintassi "...", ereditata dal \$C\$, che consente di avere un numero arbitrario di parametri per una funzione).

## Corrispondenze "esatte"

Le corrispondenze esatte corrispondono ad alcune conversioni implicite di tipo che sono garantite preservare il valore dell'argomento.

Vi sono comunque conversioni che, pur preservando il valore, non fanno parte delle corrispondenze esatte.

Le corrispondenze esatte si possono suddividere nelle seguenti sottoclassi:

### 1a. Identità (match perfetti)

In realtà, questa *NON* è una conversione, perché tipo di partenza e tipo di destinazione coincidono; è comunque comodo includerla come caso speciale nella classificazione, per poter ragionare in modo più semplice durante la risoluzione dell'overloading.

Esempio: (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

### 1b. Trasformazioni di lvalue

Esempio: (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)

### 1c. Conversioni di qualificazione

Viene aggiunto il qualificatore `const`. Esempio:

#### tipo parametro argomento

const int&	i
const int*	&i

# Le promozioni

Le promozioni corrispondono ad alcune conversioni implicite di tipo che, come le conversioni esatte, sono garantite preservare il valore dell'argomento.

Ogni implementazione del linguaggio C++ è specifica per una particolare architettura del processore (per esempio, con "parole" di 16, 32 o 64 bit). Tradizionalmente, i tipi `int` e `unsigned int` vengono forniti della dimensione adeguata per ottenere la massima efficienza su quella particolare architettura. Al contrario, i tipi interi più piccoli di `int` non sono direttamente rappresentabili all'interno del processore (per esempio, il processore sa effettuare la somma di due registri o farne il confronto, ma tipicamente non è dotato di istruzioni che effettuano la somma o effettuano il confronto tra porzioni di registri).

Quindi, ogni volta che si vuole operare su un valore di un tipo di dato più piccolo di `int`, questo deve essere "promosso" al tipo `int` (o `unsigned int`) per potere effettuare l'operazione.

## 2a. Promozioni intere

- dai tipi interi piccoli (`char/short`, `signed` o `unsigned`) al tipo `int` (`signed` o `unsigned`);
- da `bool` a `int` è promozione (caso speciale).

## 2b. Promozioni floating point

- da `float` a `double`

## 2c. Promozione delle costanti di enumerazioni del C++03

Al più piccolo tipo intero (almeno `int`) sufficientemente grande per contenerle.

[Torna all'indice](#)

---

# Conversioni standard

Le conversioni standard sono tutte le altre conversioni implicite che non coinvolgono conversioni definite dall'utente.

Attenzione: da `int` a `long` non è promozione, da `char` a `double` è conversione standard.

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_t`) sono le costanti puntatore nullo; esse possono essere convertite 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 vada 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 dato, 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 conversione implicita per il "**down-cast**", che trasformerebbe un `B*` in un `D*` o un `B&` in un `D&`).

[Torna all'indice](#)

---

# Conversioni implicite definite dall'utente

1. Uso (implicito) di costruttori che possono essere invocati con un solo argomento (di tipo diverso) e non sono marcati `explicit` Esempio:

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

2. Uso di operatori di conversione da tipo utente verso altro tipo Esempio:

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

[Torna all'indice](#)

# Gestione delle risorse

---

Una problematica rilevante quando si sviluppa software è quella di riuscire ad ottenere una corretta gestione delle risorse.

Con il termine "risorse" indichiamo genericamente entità che, intuitivamente, sono disponibili in quantità limitata e che, in caso di esaurimento, potrebbero compromettere o limitare la funzionalità del software. Per questo motivo, il software deve necessariamente interagire con le risorse in modo corretto, evitando che alcune di esse vadano "perse" o siano compromesse rendendole inutilizzabili.

In linea di massima, 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, perché le risorse disponibili sono limitate

È molto importante non utilizzare una risorsa prima di averla acquisita e dopo averla rilasciata.

In particolare:

- la risorsa deve essere acquisita prima di essere usata (o rilasciata);
- al termine del suo utilizzo la risorsa deve essere rilasciata;
- non è lecito usare una risorsa dopo averla rilasciata.

[Torna all'indice](#)

---

## Esempi di risorse

### La memoria ad allocazione dinamica

Viene acquisita tramite l'uso dell'espressione **new** (magari effettuato indirettamente), utilizzata per leggere e scrivere valori (mediante dereferenziazione di un puntatore) e infine rilasciata mediante l'uso della **delete** (magari effettuato indirettamente).

Il mancato rilascio genera **memory leak** e, in casi particolari, può portare all'esaurimento della memoria disponibile. L'accesso a **dangling pointer** è un esempio di uso dopo il rilascio. La **double free** è un esempio di rilascio di una risorsa che non è più sotto il nostro controllo (già rilasciata in precedenza).

Il termine "memory leak", che vuol dire "perdita o fuoriuscita di memoria", indica un particolare consumo non voluto di memoria causato dalla mancata deallocazione di variabili/dati non più necessari.

[Torna all'indice](#)

---

### I (descrittori dei) file del file system

Vengono acquisiti con l'operazione di apertura del file, utilizzati per leggere e/o scrivere sul file e infine rilasciati con l'operazione di chiusura. In casi specifici, la mancata operazione di chiusura di un file potrebbe comprometterne il contenuto.

[Torna all'indice](#)

---

### I lock per l'accesso (condiviso o esclusivo) a risorse condivise

L'acquisizione dei lock è necessaria per una corretta gestione della condivisione delle risorse (da parte di più processi o thread); il mancato rilascio di un lock può causare deadlock o starvation.

Esempio: le connessioni di rete a server (e.g., sessioni DBMS).

Per semplicità di esposizione, useremo spesso il caso della memoria dinamica (perché è quello più semplice da simulare negli esempi), ma deve essere chiaro che il discorso vale in generale.

[Torna all'indice](#)

---

## Exception safety

Una gestione corretta delle risorse è tutto sommato semplice da ottenere quando "tutto fila liscio", cioè quando il nostro software segue uno dei flussi di esecuzione previsti dal programmatore.

La situazione tende però a complicarsi non appena si ammette la possibilità che qualcosa possa "andare storto", ovvero quando alcune delle operazioni eseguite possono incorrere in situazioni di errore che, pur essendo state previste in linea di principio, non sono state (o non possono essere) gestite esplicitamente.

Nel caso del C++, la tecnica idiomatica per segnalare situazioni di errore consiste nel lanciare **eccezioni**, facendo quindi in modo che il programma esca dal normale flusso di esecuzione e entri nei cosiddetti flussi di esecuzione "eccezionali".

Quando eseguite in modalità eccezionale, quasi tutte le strutture di controllo del programma (concatenazione, costrutti condizionali, costrutti iterativi, ecc.) si comportano in maniera diversa dal normale, tipicamente ignorando larga parte del codice scritto dal programmatore.

Una volta lanciata un'eccezione, questa si propaga lungo la catena delle chiamate e l'unico modo di rientrare nel flusso di esecuzione normale è di catturare l'eccezione (in un blocco `catch`) e gestirla; in assenza di un blocco `catch` adeguato, il programma giungerà a terminazione in modalità eccezionale.

Diventa quindi essenziale capire nel dettaglio cosa succede in questi casi; in particolare, occorre acquisire le tecniche di programmazione che consentono di ottenere una corretta gestione delle risorse anche in presenza di comportamenti eccezionali del codice.

Questo è l'obiettivo della cosiddetta ***exception safety***.

Un esempio banale:

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

La funzione `foo` acquisisce una risorsa (la memoria dinamica puntata dal puntatore `pi`), la passa alla funzione `do_the_job` (che la usa) e infine la rilascia mediante la `delete`. In condizioni normali, `foo` gestisce correttamente la risorsa in questione, perché le tre fasi (acquisizione, uso, rilascio) si susseguono secondo l'ordine corretto.

La funzione `foo`, però, non è exception safe: se la funzione `do_the_job` (o una qualunque delle funzioni invocate da questa) lancia una eccezione e tale eccezione non viene gestita internamente, il flusso di esecuzione in uscita dalla chiamata è un flusso eccezionale, che NON esegue l'istruzione `delete pi`. Quindi, si uscirebbe dall'esecuzione di `foo` (in modalità eccezionale) senza avere rilasciato la risorsa, ottenendo un *memory leak*.

Dal punto di vista **metodologico**, la domanda che ci dobbiamo fare è quindi la seguente: quali sono le **tecniche** che possiamo utilizzare per modificare il codice della funzione `foo` affinché diventi exception safe? Tra le varie tecniche utilizzabili, ve ne sono alcune migliori di altre che, pertanto, sarebbe raccomandato seguire?

**NOTA BENE:** esistono contesti nei quali è perfettamente accettabile scrivere codice che non sia exception safe, ovvero codice che NON gestisce correttamente le risorse in presenza di comportamenti eccezionali.

Per esempio, questo capita quando:

1. La risorsa in questione è poco importante.
2. La correttezza del software (nei casi in cui si presenta un comportamento eccezionale) è di scarso interesse.
3. Il tempo di esecuzione del software è molto breve.

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

[Torna all'indice](#)

# Cosa stampa questo programma?

```
/*
Lo scopo di questo esercizio è quello di verificare la conoscenza
di alcune nozioni la cui comprensione è essenziale per affrontare
(in maniera consapevole) le tematiche relative alla corretta gestione
delle risorse e, in particolare, della exception safety.

Le nozioni in questione sono:
- tempo di vita degli oggetti
- costruzione e distruzione di oggetti
- composizione di tipi di dato (aggregazione e ereditarietà)
- flussi di esecuzione eccezionali

L'esercizio consiste nel *prevedere* l'output prodotto dal programma
(indicando con precisione la sequenza corretta dei vari messaggi).
Sebbene sia possibile (e facile) compilare ed eseguire il codice per
*vedere* l'output, il consiglio è quello di farlo soltanto in un secondo
momento, come utile feedback per verificare se le proprie previsioni
erano accurate e, nel caso, chiedersi come mai non lo erano.
Tipicamente, si scoprirà di essere incappati in qualche banale svista,
ma in alcuni casi occorrerà ammettere che certi meccanismi di base
non erano stati compresi a fondo.

*/
#include

// ----

struct C1 {
    int i1;
    C1() {
        std::cerr << "Constructor C1::C1()" << std::endl;
    }
    ~C1() {
        std::cerr << "Destructor C1::~C1()" << std::endl;
    }
};

// ----

struct C2 {
    int i2;
    C2() {
        std::cerr << "Constructor C2::C2()" << std::endl;
        throw 123;
    }
    ~C2() {
        std::cerr << "Destructor C2::~C2()" << std::endl;
    }
};

// ----

struct C3 : public C1 {
    int i3;
    C3() {
        std::cerr << "Constructor C3::C3()" << std::endl;
    }
    ~C3() {
        std::cerr << "Destructor C3::~C3()" << std::endl;
    }
};

// Memory layout per il tipo C3: { C1:{ i1 }, i3 }

// ----

class D : public C1 {
private:
    int i4;
    C3 c3;
    C2 c2;
    C1 c1;

public:
```

```

D() : c1(), c2(), c3() {
    std::cerr << "Constructor D::D()" << std::endl;
}
~D() {
    std::cerr << "Destructor D::~D()" << std::endl;
}
};

// Memory layout per il tip D:
// { C1:{ i1 }, i4, c3:C3{ C1:{ i1 }, i3 }, c2:C2{ i2 }, c1:C1{ i1 } }

// -----
C3 c3;

// -----


int main() {
    std::cout << "Start" << std::endl;
    try {
        C1 c1;
        D d;
    }
    catch (char c) {
        std::cerr << "char " << c << std::endl;
    }
    catch (...) {
        std::cerr << "..." << std::endl;
    }
    std::cout << "End" << std::endl;
    return 0;
}

// -----
C1 c1;

// -----

```

## Soluzione

```

Constructor C1::C1()
Constructor C3::C3()
Constructor C1::C1()
Start
Constructor C1::C1()
Constructor C1::C1()
Constructor C1::C1()
Constructor C3::C3()
Constructor C2::C2()
Destructor C3::~C3()
Destructor C1::~C1()
Destructor C1::~C1()
Destructor C1::~C1()
...
End
Destructor C1::~C1()
Destructor C3::~C3()
Destructor C1::~C1()

```

# Exception Safety

---

Una porzione di codice si dice **exception safe** quando si comporta in maniera "adeguata" anche in presenza di comportamenti anomali segnalati tramite il lancio di eccezioni.

In particolare, occorre valutare se la porzione di codice, in seguito al comportamento eccezionale, non abbia compromesso lo stato del programma in maniera irreparabile: esempi di compromissione sono il mancato rilascio (cioè la perdita) di risorse oppure la corruzione dello stato interno di una risorsa (ad esempio, l'invariante di classe non è più verificata), con la conseguenza che qualunque ulteriore tentativo di interagire con la risorsa si risolve in un comportamento non definito (*undefined behavior*).

---

## Livelli di exception safety

Esistono tre diversi livelli di exception safety:

- [base](#)
  - [forte](#)
  - [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 (resource leak).
2. Si è neutrali rispetto alle eccezioni quando, ogni qual volta viene ricevuta un'eccezione questa viene catturata momentaneamente, gestita in modo "locale", e successivamente viene rilasciata al chiamante (permettendo così la sua **propagazione** così che possa prenderne atto ed eseguire a sua volta eventuali azioni correttive necessarie).
3. Anche in caso di uscita in modalità eccezionale, gli oggetti sui quali si stava lavorando sono distruggibili senza causare comportamenti non definiti. Quindi lo stato interno di un oggetto, anche se parzialmente inconsistente, deve comunque consentirne la corretta distruzione (o riassegnamento).

Questo è il **livello minimo** che deve essere garantito per poter parlare di exception safety.

Gli altri livelli, che forniscono garanzie maggiori, sono spesso considerati opzionali (perché più costosi da ottenere).

[Torna all'indice](#)

---

### Livello forte

Il **livello forte (strong)** di exception safety si ottiene quando, oltre a tutte le garanzie fornite dal **livello base**, si aggiunge come ulteriore garanzia una sorta di **atomicità** delle operazioni (tutto o niente). Intuitivamente, l'invocazione di una funzione exception safe forte, in caso di eccezione, garantisce che lo stato degli oggetti manipolati è rimasto inalterato, identico allo stato precedente la chiamata.

es: *Rollback* dei DBMS.

### Esempio

Supponiamo di avere una classe che implementa una collezione ordinata di oggetti e di avere un metodo `insert` che inserisce un nuovo oggetto nella collezione esistente.

Se il metodo in questione garantisce l'exception safety forte, allora in seguito ad una eccezione durante una operazione di `insert` la collezione si troverà esattamente nello stesso stato precedente all'operazione di `insert` (cioè, conterrà

esattamente gli stessi elementi che conteneva prima della chiamata alla insert).

Se invece fosse garantita solo l'exception safety a livello base, non avendo la proprietà di atomicità, in caso di uscita con eccezione la collezione si troverebbe in uno stato consistente, ma potenzialmente in nessun rapporto con lo stato precedente alla chiamata (per esempio, potrebbe essere vuota o contenere elementi diversi rispetto a quelli contenuti precedentemente).

[Torna all'indice](#)

---

## Livello nothrow

È il **livello massimo**: una funzione è nothrow se la sua esecuzione è **garantita**, non terminare in modalità eccezionale.

Questo livello lo si raggiunge in un numero limitato di casi:

- Quando l'operazione è così semplice che **non** c'è alcuna 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 nothrow 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).

Si noti che il livello nothrow, per definizione, *NON* è neutrale rispetto alle eccezioni.

[Torna all'indice](#)

---

## Libreria standard e exception safety

I contenitori (`vector`, `deque`, `list`, `set`, `map`, ...) forniti dalla libreria standard sono **exception safe**.

Tale affermazione vale sotto determinate condizioni. Dato che si parla di contenitori templatici, quindi possono essere istanziati a partire da un qualunque tipo di dato `T`, le garanzie di exception safety del contenitore sono valide a condizione che il tipo di dato `T` degli elementi contenuti fornisca analoghe garanzie.

Molte operazioni su questi contenitori forniscono la garanzia forte (strong exception safety). Alcune però forniscono solo una garanzia base, tipicamente quando si opera su molti elementi contemporaneamente, perché quella strong sarebbe troppo costosa.

---

## Esempio strong safety

Se viene invocato il metodo `void push_back(const T& t)` su di un oggetto di tipo `std::vector<T>` e il tentativo di copiare l'oggetto `t` all'interno del `vector` dovesse fallire lanciando una eccezione (per esempio, perché il costruttore di copia di `T` ha esaurito le risorse a disposizione e non può effettuare la copia), si può essere sicuri che il `vector` *NON* è stato modificato. Se prima della chiamata conteneva gli `n` elementi `[t1, ..., tn]`, in uscita dalla chiamata contiene ancora gli stessi elementi (nello stesso ordine).

---

## Esempio base safety

Il metodo `void assign(size_type n, const T& val)` sostituisce il contenuto del vector con \$n\$ copie del valore `val`, siccome un'eccezione potrebbe essere lanciata da una qualunque delle \$n\$ operazioni di costruzione, il *vector*, in caso di eccezione, rimane in uno stato valido, ma il suo contenuto non è predicibile (in particolare, molto probabilmente il contenuto precedente è irrecuperabile).

[Torna all'indice](#)

---

## Approcci alternativi per gestire le risorse

Un esempio su tre approcci possibili che l'utente può adottare per ottenere un uso corretto di una risorsa anche in presenza di segnalazioni di errore:

- [Errori tradizionali \(no eccezioni\)](#)
  - [Uso di blocchi try/catch](#)
  - [Uso dell'idioma RAII-RRID](#)
- 

### Errori "tradizionali" (no eccezioni)

*risorsa\_no\_exc.hh*

```
#ifndef GUARDIA_risorsa_no_exc_hh
#define GUARDIA_risorsa_no_exc_hh 1

// Tipo dichiarato ma non definito (per puntatori "opachi")
struct Risorsa;

// Restituisce un puntatore nullo se l'acquisizione fallisce.
Risorsa* acquisisci_risorsa();

// Restituisce true se si è verificato un problema.
bool usa_risorsa(Risorsa* r);

// Restituisce true se si è verificato un problema.
bool usa_risorse(Risorsa* r1, Risorsa* r2);

void restituisci_risorsa(Risorsa* r);

#endif // GUARDIA_risorsa_no_exc_hh
```

*user\_no\_exc.cc*

```
#include "risorsa_no_exc.hh"

bool codice_utente() {
    Risorsa* r1 = acquisisci_risorsa();

    if (r1 == nullptr) {
        // errore durante acquisizione di r1: non devo rilasciare nulla
        return true;
    }

    // acquisita r1: devo ricordarmi di rilasciarla

    if (usa_risorsa(r1)) {
        // errore durante l'uso: rilascio r1
        restituisci_risorsa(r1);
        return true;
    }

    Risorsa* r2 = acquisisci_risorsa();

    if (r2 == nullptr) {
        // errore durante acquisizione di r2: rilascio di r1
        restituisci_risorsa(r1);
        return true;
    }

    // acquisita r2: devo ricordarmi di rilasciare r2 e r1
```

```

if (usa_risorse(r1, r2)) {
    // errore durante l'uso: rilascio r2 e r1
    restituisci_risorsa(r2);
    restituisci_risorsa(r1);
    return true;
}

// fine uso di r2: la rilascio
restituisci_risorsa(r2);
// ho ancora r1: devo ricordarmi di rilasciarla

Risorsa* r3 = acquisisci_risorsa();

if (r3 == nullptr) {
    // errore durante acquisizione di r3: rilascio di r1
    restituisci_risorsa(r1);
    return true;
}

// acquisita r3: devo ricordarmi di rilasciare r3 e r1

if (usa_risorse(r1, r3)) {
    // errore durante l'uso: rilascio r3 e r1
    restituisci_risorsa(r3);
    restituisci_risorsa(r1);
    return true;
}

// fine uso di r3 e r1: le rilascio
restituisci_risorsa(r3);
restituisci_risorsa(r1);

// Tutto ok: lo segnalo ritornando false
return false;
}

```

[Torna all'indice](#)

---

## Uso di blocchi try/catch

*risorsa\_exc.hh*

```

#ifndef GUARDIA_risorsa_exc_hh
#define GUARDIA_risorsa_exc_hh 1

#include "risorsa_no_exc.hh"

struct exception_acq_risorsa {};
struct exception_uso_risorsa {};

// Lancia una eccezione se non riesce ad acquisire la risorsa.
inline Risorsa*
acquisisci_risorsa_exc() {
    Risorsa* r = acquisisci_risorsa();
    if (r == nullptr)
        throw exception_acq_risorsa();
    return r;
}

// Lancia una eccezione se si è verificato un problema.
inline void
usa_risorsa_exc(Risorsa* r) {
    if (usa_risorsa(r))
        throw exception_uso_risorsa();
}

// Lancia una eccezione se si è verificato un problema.
inline void
usa_risorse_exc(Risorsa* r1, Risorsa* r2) {
    if (usa_risorse(r1, r2))
        throw exception_uso_risorsa();
}

#endif // GUARDIA_risorsa_exc_hh

```

## *user\_try\_catch.cc*

```
#include "risorsa_exc.hh"

void codice_utente() {
    Risorsa* r1 = acquisisci_risorsa_exc();

    try { // blocco try che protegge la risorsa r1
        usa_risorsa_exc(r1);

        Risorsa* r2 = acquisisci_risorsa_exc();

        try { // blocco try che protegge la risorsa r2
            usa_risorse_exc(r1, r2);
            restituisci_risorsa(r2);
        } // fine try che protegge r2
        catch (...) {
            restituisci_risorsa(r2);
            throw;
        }

        Risorsa* r3 = acquisisci_risorsa_exc();
        try { // blocco try che protegge la risorsa r3
            usa_risorse_exc(r1, r3);
            restituisci_risorsa(r3);
        } // fine try che protegge r3
        catch (...) {
            restituisci_risorsa(r3);
            throw;
        }
        restituisci_risorsa(r1);
    } // fine try che protegge r1
    catch (...) {
        restituisci_risorsa(r1);
        throw;
    }
}
```

Osservazioni:

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'è nulla da rilasciare).
3. La responsabilità del blocco try/catch è 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 catch usa \$\cdots\$ per catturare qualunque eccezione: non ci interessa sapere che errore si è verificato (non è nostro compito), dobbiamo solo rilasciare la risorsa protetta.
6. Nella clausola catch, dobbiamo fare due operazioni:
  - rilasciare la risorsa protetta;
  - rilanciare l'eccezione catturata (senza modificarla) usando l'istruzione `throw;`.

Il rilancio dell'eccezione catturata garantisce la "neutralità rispetto alle eccezioni": i blocchi catch catturano le eccezioni solo temporaneamente, lasciandole poi proseguire. In questo modo anche gli altri blocchi catch potranno fare i loro rilasci di risorse e l'utente otterrà comunque l'eccezione, con le informazioni annesse, potendo quindi decidere come "gestirla".

[Torna all'indice](#)

---

## Uso dell'idioma RAII-RRID

- RAII: Resource Acquisition Is Initialization
- RRID: Resource Release Is Destruction

## *risorsa\_raii.hh*

```
#ifndef GUARDIA_risorsa_raii_hh
#define GUARDIA_risorsa_raii_hh 1
```

```

#include "risorsa_exc.hh"

// classe RAII-RRID (spesso detta solo RAII, per brevità)
// RAII: Resource Acquisition Is Initialization
// RRID: Resource Release Is Destruction

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

    // Distruttore: rilascia la risorsa (RRID)
    ~Gestore_Risorsa() {
        // Nota: si assume che restituisci_risorsa si comporti correttamente
        // quando l'argomento è il puntatore nullo; se questo non è il caso,
        // è sufficiente aggiungere un test prima dell'invocazione.
        restituisci_risorsa(res_ptr);
    }

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

    // 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;
    }

    // Accessori per l'uso (const e non-const)
    const Risorsa* get() const { return res_ptr; }
    Risorsa* get() { return res_ptr; }

    // Alternativa agli accessori: operatori di conversione implicita
    // operator Risorsa*() { return res_ptr; }
    // operator const Risorsa*() const { return res_ptr; }

}; // class Gestore_Risorsa

#endif // GUARDIA_risorsa_raii_hh

```

### *user\_raii.cc*

```

#include "risorsa_raii.hh"

void codice_utente() {
    Gestore_Risorsa r1;
    usa_risorsa_exc(r1.get());
{
    Gestore_Risorsa r2;
    usa_risorse_exc(r1.get(), r2.get());
} // L'inserimento di questo blocco serve per fare in modo che
  // lo scope di r2 finisca proprio in questo punto,
  // prima di inizializzare r3
    Gestore_Risorsa r3;
    usa_risorse_exc(r1.get(), r3.get());
}

```

[Torna all'indice](#)

---

## Esercizio

![[eccezioni\_esempio.jpg]]

Soluzione:

```
void job() {
    Res* r1 = new Res("res1");
    try {
        Res* r2 = new Res("res2");
        try {
            do_task(r1, r2);
            delete res1;
            delete res2;
        } catch (...) {
            delete res2;
            throw;
        }
    } catch (...) {
        delete res1;
        throw;
    }
}
```

[Torna all'indice](#)

---

## Altro materiale

- Articolo di Stroustrup Exception Safety: Concepts and Techniques <http://www.stroustrup.com/except.pdf>
- Video e lucidi della presentazione di Jon Kalb al CppCon 2014 (con bonus per i fan di Star Wars) Video parte 1: [https://www.youtube.com/watch?v=W7fly\\_54y-w](https://www.youtube.com/watch?v=W7fly_54y-w) Video parte 2: <https://www.youtube.com/watch?v=b9xMIKb1jMk> Video parte 3: <https://www.youtube.com/watch?v=MiKxfdkMJW8> Ludici: <http://exceptionsafecode.com/slides/esc.pdf>

[Torna all'indice](#)

# Smart pointers

---

Come accennato quando si è introdotto il discorso della gestione delle risorse e dell'exception safety, uno dei casi più frequenti che si verificano è quello della corretta gestione dell'allocazione dinamica della memoria.

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

L'idioma RAII-RRID si presta bene a neutralizzare la maggior parte di questi errori, rendendoli molto meno probabili. D'altra parte, scrivere una classe RAII per ogni tipo  $T$  ogni volta che si vuole usare un  $T^*$  è operazione noiosa, ripetitiva e comunque soggetta a errori.

La libreria standard viene però in aiuto, fornendo delle classi tempiatiche che forniscono diverse tipologie di puntatori "smart": `unique_ptr`, `shared_ptr` e `weak_ptr`. Le tre classi tempiatiche sono definite nell'header file `<memory>`.

**NOTA BENE:** 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).

[Torna all'indice](#)

---

## unique\_ptr

Uno `std::unique_ptr<T>` è 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, allo smart pointer spetta l'onere di fornire una corretta gestione della risorsa (nello specifico, rilasciarla a lavoro finito).

Esempio:

```
#include

void foo() {
    std::unique_ptr pi(new int(42));
    std::unique_ptr 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 rilasciata la memoria
```

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

Esempio:

```
void foo(std::unique_ptr pi);

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

La classe fornisce poi metodi per potere 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 pi;      // pi non gestisce (ancora) una risorsa
int* raw_pi = new int(42);

pi.reset(raw_pi);        // NON devo invocare la delete su raw_pi

int* raw_pj = pi.get();    // NON devo invocare la delete su raw_pj
int* raw_pk = pi.release(); // devo invocare la delete su raw_pk

```

Con il metodo `reset()` il puntatore prende in gestione una nuova risorsa (diventandone il proprietario), rilasciando la risorsa che aveva in gestione precedentemente, se presente.

Il metodo `get()` fornisce il puntatore raw alla risorsa gestita, che però rimane sotto la responsabilità dello `unique_ptr`; il metodo `release()`, invece, restituisce il puntatore raw e ne cede anche la responsabilità di corretta gestione.

[Torna all'indice](#)

---

## shared\_ptr

Uno `std::shared_ptr<T>` è un puntatore smart ad un oggetto di tipo `T`. Lo `shared_ptr` implementa il concetto di puntatore per il quale la responsabilità della corretta gestione della risorsa è "condivisa": intuitivamente, ogni volta che uno `shared_ptr` viene copiato, l'originale e la copia condividono la responsabilità della gestione della (stessa) risorsa.

A livello di implementazione, la copia causa l'incremento di un contatore del numero di riferimenti 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 avervi ancora accesso, ne effettua il rilascio (informalmente, si dice che "*l'ultimo chiude la porta*").

Esempio:

```

#include

void foo() {
    std::shared_ptr pi;

    {
        std::shared_ptr pj(new int(42)); // ref counter = 1
        pi = pj; // condivisione risorsa, ref counter = 2
        ++(*pi); // 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 pj, ref counter = 0, rilascio risorsa

```

Come detto, gli `shared_ptr` sono copiabili (e spostabili). La classe fornisce i metodi `reset()` e `get()`, con la semantica intuitiva.

Esempio:

```

void foo(std::shared_ptr pi);

void bar() {
    std::shared_ptr pj(new int(42));

    foo(pj); // ok: copia ammessa, risorsa condivisa
    std::cout << *pj; // stampa la risorsa come modificata da foo

    foo(std::move(pj)); // ok: spostamento ammesso
    // dopo lo spostamento, pj non gestisce nessuna risorsa
}

```

[Torna all'indice](#)

---

# Template di funzione (make\_shared e 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 è la garanzia offerta dalla `std::make_shared`.

Esempio:

```
void bar() {
    auto pi = std::make_shared(42);
    auto pj = std::make_shared(3.1415);
}
```

Oltre all'efficienza, l'uso di `std::make_shared` consente di evitare alcuni errori subdoli che potrebbero compromettere la corretta gestione delle risorse in presenza di comportamenti eccezionali.

Esempio:

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

void foo() {
    // codice NON exception safe
    bar(std::shared_ptr(new int(42)),
        std::shared_ptr(new int(42)));

    // codice exception safe
    bar(std::make_shared(42),
        std::make_shared(42));
}
```

Siccome l'ordine di esecuzione delle sottoespressioni è non specificato, nella prima chiamata della funzione `bar()` una implementazione potrebbe decidere di valutare per prime le due espressioni `new` passate come argomenti ai costruttori degli `shared_ptr` e solo dopo invocare i costruttori.

Se la prima allocazione tramite `new` andasse a buon fine ma la seconda invece fallisse con una eccezione, si otterebbe un memory leak (per la prima risorsa allocata), in quanto il distruttore dello `shared_ptr` NON verrebbe invocato (perché l'oggetto non è stato costruito).

Il problema non si presenta nella seconda chiamata a `bar()`, perché le allocazioni sono effettuate (implicitamente) dalla `make_shared`.

**NOTA:** questo esempio NON dovrebbe causare un problema di exception safety nel caso di una implementazione conforme allo standard C++17: in questo standard, infatti, è stata modificata la regola relativa all'ordine di valutazione degli argomenti in una chiamata di funzione.

A partire dallo standard C++14 è stata resa disponibile anche la `std::make_unique`. L'uso degli smart pointer e di queste funzioni per la loro creazione dovrebbe consentire al programmatore di limitare al massimo la necessità di utilizzare (esplicitamente) le espressioni `new` e le corrispondenti invocazioni di `delete`: in effetti, nelle più recenti linee guida alla programmazione in C++, l'uso diretto (naked) di `new` e `delete` è considerato "cattivo stile", quasi quanto l'uso dell'istruzione `goto`.

<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

[Torna all'indice](#)

---

## weak\_ptr

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

In questo caso, le risorse comprese in un ciclo mantengono dei reference count positivi anche se non sono più raggiungibili a partire dagli `shared_ptr` ancora accessibili da parte del programma, causando dei memory leak. L'uso dei `std::weak_ptr` è pensato per risolvere questi problemi.

Un `weak_ptr` è un puntatore ad 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 la indirizzano. Ciò significa che un `weak_ptr` non può accedere direttamente alla risorsa: prima di farlo, deve controllare se la risorsa è ancora disponibile. Il modo migliore per farlo è mediante l'invocazione del metodo `lock()`, che produce uno `shared_ptr` a partire dal `weak_ptr`: se la risorsa non è più disponibile, lo `shared_ptr` ottenuto conterrà il puntatore nullo.

Esempio:

```
void maybe_print(std::weak_ptr wp) {
    if (auto sp2 = wp.lock())
        std::cout << *sp2;
    else
        std::cout << "non più disponibile";
}

void foo() {
    std::weak_ptr wp;
    {
        auto sp = std::make_shared(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 più disponibile"
}
```

[Torna all'indice](#)

# Template in C++

---

## Template di funzione

Un template di funzione è un costrutto del linguaggio C++ che consente di scrivere un *modello* (schema) parametrico per una funzione.

Esempio:

```
// dichiarazione pura di un template di funzione
template
T max(T a, T b);

// definizione di un template di funzione
template
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Nell'esempio precedente abbiamo prima dichiarato e poi definito un template di funzione di nome `max`.

Come al solito, la definizione è anche una dichiarazione.

Nell'esempio, `T` è un parametro di template: il parametro viene dichiarato essere un `typename`(nome di tipo) nella lista dei parametri del template. La parola chiave "`typename`" può essere sostituita da "`class`", ma in ogni caso indica un qualunque tipo di dato, anche built-in (quindi per coerenza si dovrebbe preferire l'uso di "`typename`"). La lista dei parametri può contenerne un numero arbitrario, separati da virgolette; oltre ai parametri che sono nomi di tipo, vedremo che esistono anche altre tipologie (valori, template).

Si noti che per convenzione (non è una regola del linguaggio) si usano nomi maiuscoli per i parametri di tipo; il nome "`T`" è comunque arbitrario (e può essere cambiato a piacere): è stato scelto, probabilmente, per indicare che si intende un tipo qualunque.

Esempio:

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

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

[Torna all'indice](#)

---

## Istanziazione di un template di funzione

Dato un template di funzione, è possibile "generare" da esso una o più funzioni mediante il meccanismo di istanziazione (del template): l'istanziazione fornisce un *argomento* (della tipologia corretta) ad ognuno dei parametri del template.

L'istanziazione avviene spesso in maniera *implicita*, quando si fa riferimento al nome del template allo scopo di "usarne" una particolare istanza. Nell'esempio seguente, il template `max` viene istanziato (implicitamente) due volte, usando le parentesi angolate per fornire (esplicitamente) l'argomento per il parametro del template.

Esempio:

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

    // istanziazione della funzione
```

```
// double max(double, double)
double d = max(d1, d2);
}
```

Quando si istanzia un template di funzione, è possibile evitare la sintassi 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) {
    // istanziazione della funzione
    // char max(char, char);
    int m = max(c1, c2);
    // il legame T = char viene dedotto dal tipo degli argomenti c1 e c2;
    // si noti che il tipo di m (int) non influisce sul processo di deduzione
}
```

Il processo di deduzione potrebbe fallire a causa di ambiguità:

```
void foo(double d, int i) {
    int m = max(d, i); // errore
    // il compilatore non può dedurre un unico tipo T coerente con d e i
    int m = max(d, i); // ok: evito la deduzione
}
```

[Torna all'indice](#)

---

## Nota bene

È opportuno sottolineare la differenza sostanziale tra un template di funzione e le sue possibili istanziazioni. In particolare: un template di funzione NON è una funzione (è un "generatore" di funzioni); una istanza di un template di funzione è una funzione.

Per esempio, 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).

Se compilo una unità di traduzione ottenendo un object file e osservo il contenuto dell'object file con il comando `nm`, vedrò solo le *istanze* dei template di funzione (non vedrò i template di funzione).

Fatta questa doverosa sottolineatura, va comunque detto che nel linguaggio comune spesso si parla di "chiamata di un template di funzione" per indicare la chiamata di una sua specifica istanza.

[Torna all'indice](#)

---

## Una utile analogia

A livello intuitivo può essere utile fare la seguente analogia. Sui siti web dei corsi di laurea sono spesso resi disponibili dei moduli per compilare una domanda di modifica di piano di studio.

Questi moduli sono dei "modelli" di domanda, parametrici, e corrispondono al concetto di template: in essi sono lasciati degli spazi (i parametri) che devono essere compilati con i dati dello studente per produrre una (vera e propria) domanda di modifica di piano di studio.

Il processo di istanziazione corrisponde alla compilazione del modulo: ad ogni parametro si associa il corrispondente argomento. Il modulo compilato corrisponde quindi all'istanza del template. Alla segreteria studenti occorre fare avere l'istanza (il modulo compilato), in quanto del template (il modulo in bianco) non saprebbero che farsene.

[Torna all'indice](#)

---

## 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, molto probabilmente, le intenzioni dell'utente era di fare un confronto lessicografico tra due stringhe stile `$C$`.

Per ovviare, è possibile fornire una definizione alternativa della funzione templatica, "specializzata" per gli argomenti problematici, nel modo seguente:

```
// definizione del template
template
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* a, const char* b) {
    return strcmp(a, b) > 0;
}
```

A livello sintattico, si noti la lista vuota dei parametri `<>`, ad indicare che si tratta di una specializzazione *totale* (non sono ammesse specializzazioni parziali per i template di funzione).

Anche in questo caso è possibile omettere la lista degli argomenti `<const char*>`, lasciando che venga dedotta dal compilatore.

```
template <>
const char* max(const char* a, const char* b) { ... }
```

Si noti che sarebbe comunque stato possibile evitare la specializzazione del template e fornire la versione specifica per i `const char*` come funzione "normale", sfruttando l'overloading di funzioni:

```
const char* max(const char* a, const char* b);
```

Diventa quindi importante capire come si comporta il meccanismo di risoluzione dell'overloading in questi casi (l'approfondimento verrà effettuato successivamente).

[Torna all'indice](#)

---

## Instanziazioni esplicite di template

Abbiamo visto che per un template è possibile fornire istanziazioni implicite (date dall'uso del template) e specializzazioni esplicite. Esiste anche la possibilità di richiedere *esplicitamente* al compilatore l'istanziazione di un template, indipendentemente dal fatto che questo venga effettivamente utilizzato.

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

### 1. Dichiarazione di istanziazione esplicita

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

### 2. Definizione di istanziazione esplicita

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

A livello sintattico, si noti l'assenza della lista dei parametri (la parola chiave `template` NON è seguita da parentesi angolate): questo differenzia le istanziazioni esplicite dalle specializzazioni esplicite.

Il caso 1 (dichiarazione) informa il compilatore che, quando verrà usata quella istanza del template, NON deve essere prodotta la corrispondente definizione dell'istanza (evitando quindi la generazione del codice). Intuitivamente, la parola chiave `extern` indica che il codice dovrà essere trovato dal linker "esternamente" a questa unità di traduzione, cioè in un object file generato dalla compilazione di un'altra unità di traduzione. In pratica, la *dichiarazione* di istanziazione esplicita impedisce che vengano effettuate le istanziazioni implicite (per diminuire i tempi di compilazione e/o generare object file più piccoli).

Il caso 2 (definizione) è complementare al caso 1: si informa il compilatore che quella particolare istanza del template va generata, a prescindere dal fatto che in questa unità di traduzione ne venga o meno effettuato l'utilizzo. Serve ad assicurarsi che le altre unità di traduzione (che hanno usato il caso 1) possano essere collegate con successo.

[Torna all'indice](#)

---

## Template di classe

Un template di classe è un costrutto del linguaggio che consente di scrivere un *modello* parametrico per una classe.

Quasi tutti i concetti esposti per il caso dei 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
class Stack;

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

Nel caso dei template di classe, è ancora più 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 s1; // istanziazione implicita del tipo Stack
           // (in particolare, del costruttore di default)

Stack s2 = s1; // errore: non viene dedotto il tipo T = int

auto s2 = s1; // ok: il C++11 ha introdotto la deduzione di tipo
              // dall'inizializzatore, usando `auto`; viene anche
              // istanziato implicitamente il costruttore di copia
```

L'unico caso in cui è lecito usare il nome del template di classe per indicare (in realtà) il nome della classe ottenuta mediante istanziazione è all'interno dello scope del template di classe stesso.

Per esempio:

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

```
// definizione (out-of-line)
template
Stack& // il tipo di ritorno è fuori scope di classe, devo scrivere
Stack::operator=(const Stack& y) { // parametro in scope di classe
    Stack tmp = y; // in scope di classe, è sufficiente Stack
    swap(tmp);
    return *this;
}
```

[Torna all'indice](#)

---

## Istanziazione on demand

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'istanziazione. Quindi, nell'esempio precedente, per la classe `Stack<int>` NON vengono istanziati i metodi `Stack<int>::push(const int&)` e `Stack<int>::pop()`. Verranno istanziati se e quando utilizzati.

Questo scelta del linguaggio ha conseguenze positive e negative:

- In negativo: quando scrivo i test per la classe templatica devo prestare attenzione a fornire un insieme di test che copra tutte le funzionalità di interesse; le funzionalità NON testate (e quindi non istanziate) potrebbero addirittura generare errori di compilazione al momento dell'istanziazione da parte dell'utente.
- In positivo: per lo stesso motivo, posso usare un sotto-insieme delle funzionalità della classe istanziandola con argomenti che soddisfano solo i requisiti di quelle funzionalità; il fatto che quegli argomenti siano "scorretti" per le altre funzionalità (non usate) non mi impedisce l'utilizzo dell'interfaccia "ristretta".

Esempio: Supponiamo che la classe `Stack<T>` fornisca un metodo `print()`, implementato invocando il corrispondente metodo `print()` del parametro `T` su ognuno degli oggetti contenuti nello stack. Questo significa che, per usare il metodo `Stack<T>::print()`, il tipo `T` dove fornire a sua volta il metodo `T::print()`.

Si noti, per esempio, che "int" non è una classe e quindi un tentativo di istanziare `Stack<int>::print()` genera un errore di compilazione. L'errore, però, lo si ottiene *solo* se effettivamente si prova a istanziare `Stack<int>::print();`; l'istanziazione dei metodi `Stack<int>::push()` e `Stack<int>::pop()` continua ad essere corretta e utilizzabile.

[Torna all'indice](#)

---

## Istanziazioni e specializzazioni di template di classe

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

Un esempio di specializzazione *totale* di template di classe è 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 di uso:

```
#include

int foo() {
    long minimo = std::numeric_limits::min();
    long massimo = std::numeric_limits::max();
    bool char_con_segno = std::numeric_limits::is_signed;
}
```

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

```
// [...]
// numeric_limits specialization.
template<>
```

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

Un altro esempio è dato dalla specializzazione `std::vector<bool>` del template `std::vector`, creata allo scopo di fornire una versione del contenitore ottimizzata per risparmiare memoria (codificando ogni valore booleano con un singolo bit).

[Torna all'indice](#)

---

## Specializzazioni parziali

A differenza dei template di funzione, i template di classe supportano anche le *specializzazioni parziali*. Si tratta di specializzazioni di template che sono applicabili non per una scelta specifica degli argomenti (come nel caso delle specializzazioni totali), ma per sottoinsiemi di tipi. Una specializzazione parziale di un template (di classe), quindi, è ancora un template di classe, ma di applicabilità meno generale.

[Torna all'indice](#)

---

## Un'altra analogia

Riprendiamo l'analogia dei moduli per la domanda di modifica di piano di studi: si era detto che il modulo in bianco è il template (per uno studente qualsiasi) e il modulo compilato in ogni sua parte è l'istanza (di uno specifico studente).

Una specializzazione esplicita *totale* corrisponde ad una domanda di modifica di piano degli studi "fuori standard", fatta (ad personam) da uno specifico studente e che non segue necessariamente lo schema del modello generale.

Una specializzazione *parziale*, invece, corrisponde ad un modulo diverso (quindi è ancora un template), che però viene utilizzato solo da uno specifico sottoinsieme degli studenti (per esempio, gli studenti iscritti alle lauree magistrali a ciclo unico). Quando uno studente di Medicina e Chirurgia chiede il modulo da compilare, gli si fornisce il modulo specializzato (parzialmente): per ottenere la domanda vera e propria dovrà compilare (istanziare) il modulo specializzato.

Gli esempi di specializzazione parziale di template di classe sono meno frequenti (anche nella libreria standard). Tra di essi tratteremo (quando affronteremo gli iteratori) il caso della specializzazione parziale per i puntatori degli `std::iterator_traits`. Nell'header file `<iterator>` (in realtà, in un header file interno che dipende dalla specifica implementazione) troviamo:

```
// Partial specialization for pointer types.
template
struct iterator_traits<_Tp*>
/* ... */
```

Il fatto che si tratti di una specializzazione parziale si deduce dalla contemporanea presenza della lista (non vuota) dei parametri del template e della lista (non vuota) degli argomenti del template, nella quale si nomina ancora il parametro del template.

[Torna all'indice](#)

---

## Altri template

Gli standard più recenti hanno introdotto nuove forme di template, sui quali non faremo approfondimenti.

Template di alias:

```
template
using Vec = std::vector>;
```

Template di variabile:

```
template
const T pi = T(3.1415926535897932385L);
```

[Torna all'indice](#)

---

## Compilazione dei template

Il processo di compilazione dei template richiede che lo stesso codice sia analizzato dal compilatore in (almeno) due contesti distinti:

1. al momento della definizione del template, e
2. al momento della instanziazione del template.

Nella prima fase (definizione del template) il compilatore si trova ad operare con informazione incompleta. Si consideri il seguente esempio:

```
template
void incr(int& i, T& t) {
    ++i; // espressione indipendente dai parametri del template
    ++t; // espressione dipendente dai parametri del template
}
```

Sulla prima espressione il compilatore può effettuare tutti i controlli di correttezza previsti (sintattici e di semantica statica) e nulla vieterebbe di generare una porzione del codice oggetto.

Sulla seconda espressione, invece, gli unici controlli che possono essere effettuati sono dei banali controlli sintattici: non c'è modo per il compilatore di sapere se il tipo T fornisce effettivamente un operatore di preincremento. Questo controllo (e la segnalazione di eventuali errori) viene quindi "rimandato" alla successiva fase di instanziazione del template.

[Torna all'indice](#)

---

## Conseguenza 1

Una prima conseguenza, di cui tenere conto quando si scrivono i programmi, è che la **definizione di un template deve essere disponibile i tutti i punti del programma nei quali se ne richiede l'istanziazione**.

In pratica, esistono tre modi per organizzare il codice sorgente quando si scrivono funzioni o classi templatistiche:

1. Includere le definizioni dei template (comprese la definizioni di eventuali funzioni membro dei template di classe) prima di ogni loro uso nella unità di traduzione.
2. Includere le dichiarazioni del template (comprese le dichiarazioni delle eventuali funzioni membro dei template di classe) prima di farne uso e successivamente (prima o dopo gli usi) includere le definizioni del template nella unità di traduzione.
3. Sfruttando il meccanismo delle istanziazioni esplicite, includere solo le dichiarazioni dei template e le *dichiarazioni* di istanziazione esplicita prima di ogni loro uso nell'unità di traduzione, assicurandosi che le definizioni dei template e le *definizioni* di istanziazione esplicita siano fornite in un'altra unità di traduzione.

L'approccio più comune, perché più semplice, è il primo. Il secondo approccio si usa solo quando necessario (per esempio, nel caso di funzioni templatistiche che si invocano l'un l'altra ricorsivamente). Il terzo approccio è usato raramente, tipicamente al solo scopo di diminuire i tempi di compilazione.

Si noti che, nell'esercitazione sulla templatizzazione della classe Stack, abbiamo seguito il primo approccio, spostando tutte le definizioni dei metodi della classe all'interno dell'header file Stack.hh.

[Torna all'indice](#)

## Conseguenza 2

Una seconda conseguenza del meccanismo di compilazione in due fasi dei template è che, in alcuni casi, occorre modificare il codice di implementazione dei template di funzioni o classe per fornire al compilatore qualche informazione utile ad evitare errori di compilazione.

Per esempio, consideriamo il seguente codice (non templatico):

```
struct S {  
    using value_type = /* ... dettaglio implementativo ... */;  
    /* ... */  
};  
  
void foo(const S& s) {  
    S::value_type* ptr;  
    /* ... */  
}
```

Supponiamo ora di voler templatizzare la classe S, rendendola parametrica rispetto ad un qualche tipo usato al suo interno. Intuitivamente, il processo di "lifting" porterebbe ad adattare il codice in questo modo:

```
template  
struct S {  
    using value_type = /* ... dettaglio implementativo ... */;  
    /* ... */  
};  
  
template  
void foo(const S& s) {  
    S::value_type* ptr; // errore: ptr non dichiarato  
    /* ... */  
}
```

Il compilatore segnala un errore quando esamina la definizione del template di funzione foo (nella prima fase della compilazione dei template). In effetti, il compilatore si trova di fronte a codice del tipo

```
nome1 * nome2
```

e non sa nulla di nome1 e nome2: siccome nome1 è dipendente dal parametro template T, il compilatore assume che sia il nome di un "valore" (non di un "tipo"), interpretando l'istruzione come applicazione dell'operatore \* binario; viene quindi segnalato un errore perché nome2 (che non dipende dal parametro T) non è stato dichiarato.

Per risolvere il problema e comunicare correttamente le nostre intenzioni al compilatore, occorre informarlo che S<T>::value\_type indica il nome di un tipo, aggiungendo la parola chiave typename:

```
template  
void foo(const S& s) {  
    typename S::value_type* ptr; // ok, dichiaro un puntatore  
    /* ... */  
}
```

NOTA BENE: occorre rendersi conto che la problematica suddetta si potrebbe presentare in maniera subdola. Si consideri per esempio questa variante:

```
int p = 10;  
  
template  
void foo(const S& s) {  
    S::value_type* p; // compila senza errori (operator* binario)  
    /* ... */  
}
```

Per pura sfortuna, esiste una dichiarazione di un intero p visibile quando il compilatore valuta l'istruzione, per cui il compilatore non rileva l'errore (assumendo che il programmatore intenda fare una sorta di moltiplicazione del valore di S<T>::value\_type con il valore 10 memorizzato nella variabile p dello scope globale).

# 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 è "*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*", che affronteremo in un'altra parte del corso.

La programmazione generica (in \$C\$++) è una metodologia di programmazione fortemente basata sul polimorfismo statico (ovvero sui template). Sarebbe però sbagliato pensare che la programmazione generica sia semplicemente basata su definizione e uso di template di classe e funzione: i maggiori benefici della programmazione generica si ottengono solo quando un certo numero di template sono progettati in maniera coordinata, allo scopo di fornire interfacce comuni e facilmente estendibili.

Per comprendere meglio questo punto è utile studiare quella parte della libreria standard del \$C\$++ che fornisce i contenitori e gli algoritmi, cercando di capire come questi riescano ad integrarsi tra di loro.

[Torna all'indice](#)

---

## Contenitori

Un **contenitore** è 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, 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.

[Torna all'indice](#)

---

## 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 è stabilito in base ad un criterio di **ordinamento** a priori, ma viene dato dalle specifiche operazioni di inserimento e rimozione degli elementi (effettuati a partire da posizioni determinate della sequenza).

I contenitori sequenziali standard sono:

- [Vector](#)
- [Deque](#)
- [List](#)
- [Forward list](#)

[Torna all'indice](#)

---

## Vector

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

Sequenza di T di dimensione variabile (a tempo di esecuzione), memorizzati in modo contiguo. Fornisce accesso ad un qualunque elemento in tempo costante. Inserimenti e rimozioni di elementi sono (ragionevolmente) efficienti se fatti in fondo alla sequenza; altrimenti è necessario effettuare un numero lineare di spostamenti di elementi per creare (eliminare) lo spazio per effettuare l'inserimento (rimozione).

È presente un metodo per ottenere un puntatore al primo elemento della sequenza così da permettere l'integrazione con funzioni che lavorano con un puntatore ad un array.

---

## Deque

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

Una "double-ended queue" è una coda a doppia entrata, nella quale inserimenti e rimozioni efficienti possono essere effettuati sia in fondo alla sequenza (come nel caso dei vector) che all'inizio della sequenza. Per poterlo fare, si rinuncia alla garanzia di memorizzazione contigua degli elementi (intutivamente, gli elementi vengono memorizzati in "blocchi"). Fornisce accesso ad un qualunque elemento in tempo costante.

---

## List

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

Sequenza di  $T$  di dimensione variabile (a tempo di esecuzione), memorizzati (in modo non contiguo) in una struttura a lista doppiamente concatenata. La doppia concatenazione consente lo scorrimento della lista sia in avanti che all'indietro (bidirezionale). Per accedere ad un elemento occorre "raggiungerlo" seguendo i link della lista. Inserimenti e rimozioni possono essere effettuati in tempo costante (nella posizione corrente), perché non occorre spostare elementi.

---

## Forward list

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

Come una list, ma la concatenazione tra nodi è singola e quindi è consentito lo scorrimento solo in avanti (forward).

[Torna all'indice](#)

---

## Pseudo-contenitori

Oltre ai veri contenitori sequenziali, ve ne sono alcuni che sono detti "*pseudo-contenitori*":

- [Array](#)
- [String](#)
- [Bitset](#)

[Torna all'indice](#)

---

## Array

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

Sequenza di  $T$  di dimensione  $N$ , fissata a tempo di compilazione.

Nota:  $N$  è un parametro valore, non è un typename.

Intuitivamente corrisponde ad un array del linguaggio, ma è immune dalle problematiche relative al type decay e consente di conoscere facilmente il numero di elementi.

---

## String

```
std::string
```

Può essere visto come una sequenza di caratteri (char).

Nota: `std::string` è un alias per l'istanza `std::basic_string<char>` del template `std::basic_string`; il template si può istanziare con altri tipi carattere, per cui abbiamo gli alias `std::wstring`, `std::u16string` e `std::u32string` per le stringhe di `wchar_t`, `char16` e `char32`.

---

## Bitset

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

Una sequenza di esattamente `N` bit.

Nota: `N` è un parametro *valore*, non è un typename.

[Torna all'indice](#)

---

## Le operazioni sui contenitori sequenziali

I contenitori sequenziali forniscono:

- costruttori
- operatori per interrogare (gestire) la dimensione
- operatori per consentire l'accesso agli elementi
- operatori per inserire e rimuovere elementi
- operatori di confronto (tra contenitori)
- alcuni altri operatori specifici

È opportuno esaminare in dettaglio le interfacce dei vari contenitori, mettendo in evidenza le somiglianze e le differenze (e magari chiedendosi il motivo di certe differenze). Per farlo, oltre allo studio del libro di testo, è possibile consultare la corrispondente documentazione disponibile online, per esempio ai seguenti indirizzi:

- [www.en.cppreference.com](http://www.en.cppreference.com)
- [www.cppplusplus.com](http://www.cppplusplus.com)

Nota: si rimanda ad un momento successivo l'introduzione ai contenitori associativi della libreria standard.

[Torna all'indice](#)

---

## Uno sguardo a `std::vector`

La dichiarazione nel file header presenta un oggetto di tipo allocator. Questo permette di utilizzare un metodo di allocazione "personalizzata", per gli oggetti rappresentati da `T`. Se non viene specificato verrà utilizzato in automatico quello dello standard.

```
template<
    class T,
    class Allocator = std::allocator
> class vector;
```

Nota: `allocator` è comune a tutti i contenitori. Per sapere di più su [std::allocator](#).

I dati membro:

- `value_type`
- `allocator_type`
- `szye_type`
- `difference_type`
- `reference`
- `const_reference`: reference che permette l'accesso in sola lettura
- `pointer`

- `const_pointer`
- 4 tipi di iteratori (importanti):
  - `iterator`
    - `const_iterator`: permette di iterare sul vector in sola lettura;
    - `reverse_iterator`: fa il contrario di quello che gli viene detto di fare, ad esempio se gli si chiede l'inizio fornisce la fine;
    - `const_reverse_iterator`: è un `reverse_iterator` che permette la sola lettura;

All'interno dei contenitori è presente un costruttore che è considerabile un **coltellino svizzero**:

```
template
vector(InputIt first, InputIt last,
        const Allocator& alloc = Allocator());
```

Questo permette di inizializzare un contenitore iterando gli elementi compresi tra gli iteratori `first` e `last`.

Esempio:

```
int main() {
    // creo una coda
    std::queue dd;
    /*
    ... popolo dd ...
    */

    // creo un vettore sfruttando il costruttore sopra descritto, iterando tutti gli elementi della coda
    std::vector v(dd.begin(), dd.end());
    return 0;
}
```

Nota: `dd.end()` fa riferimento all'elemento successivo all'ultimo.

[Torna all'indice](#)

---

## Algoritmi generici: dai tipi ai concetti

Abbiamo brevemente introdotto quattro contenitori sequenziali standard e (almeno) tre quasi-contenitori: studiandoli in dettaglio, ci siamo probabilmente accorti che sono caratterizzati da interfacce simili, ma non esattamente identiche.

Abbiamo anche notato che le interfacce NON comprendono molti dei servizi che un utente si aspetta di potere utilizzare quando vuole lavorare con collezioni di elementi: per esempio, la classe `vector` non fornisce un metodo per ordinare gli elementi o per controllare se un elemento con un determinato valore è presente al suo interno.

L'idea di fondo della libreria standard è che questi servizi vengano implementati come "algoritmi generici", all'esterno dei contenitori e in modo il più possibile indipendente rispetto alla specifica implementazione fornita da un determinato tipo contenitore. In altre parole, gli algoritmi generici non sono pensati per lavorare con tipi di dato specifici; piuttosto, sono pensati per lavorare su "concetti" astratti ed essere quindi applicabili a tutti i tipi di dato che forniscono tutte le garanzie che caratterizzano un concetto.

Per rimanere su un esempio concreto, consideriamo un algoritmo che debba cercare un elemento con un certo valore all'interno di un contenitore. A ben pensarci, questo algoritmo non ha una vera necessità di operare su di un tipo contenitore: visto in astratto, l'algoritmo può essere applicato ad una qualunque sequenza i cui elementi possano essere scorsi, dall'inizio alla fine, e confrontati con l'elemento cercato. In altre parole, per questo algoritmo di ricerca, il tipo contenitore può essere sostituito dal "concetto" astratto di sequenza sulla quale si possano fare operazioni di lettura.

Un modo per rappresentare una **sequenza** dalla quale vogliamo leggere è quello di utilizzare una coppia di iteratori (convenzionalmente chiamati `first` e `last`), che servono ad indicare la posizione iniziale della sequenza (`first`) e la posizione subito dopo l'ultima (`last`). Si tratta quindi di sequenze "semi-aperte", spesso informalmente indicate con la notazione degli intervalli:

`$$[;first;; last;)$$`

dove la parentesi quadra iniziale ci informa che l'elemento indicato da first è compreso nella sequenza, mentre la parentesi tonda finale ci informa che l'elemento indicato da last è escluso dalla sequenza.

[Torna all'indice](#)

---

## Che cosa è un iteratore?

Un iteratore non è un tipo di dato; è un "conceitto" astratto (come il concetto di sequenza).

L'esempio classico di iteratore è dato dal tipo puntatore ad un elemento contenuto in un array: usando una coppia di puntatori, posso identificare una porzione dell'array come la sequenza sulla quale applicare un algoritmo. Il primo puntatore punta al primo elemento della sequenza, il secondo puntatore punta alla posizione immediatamente successiva all'ultimo elemento della sequenza che voglio esaminare.

Esempio che utilizza un iteratore concreto (`int*`):

```
int* cerca(int* first, int* last, int elem) {
    for ( ; first != last; ++first)
        if (*first == elem)
            return first;
    return last;
}

int main() {
    int ai[200] = { 1, 2, 3, 4, ... };
    int* first = ai;
    int* last = ai + 2; // cerco solo nei primi 3 elementi
    int* ptr = cerca(first, last, 2);
    if (ptr == last)
        std::cerr << "Non trovato";
    else
        std::cerr << "Trovato";
}
```

L'algoritmo di ricerca mostrato sopra funziona solo per i puntatori a interi; per aumentarne l'applicabilità, dovremmo fare la templatizzazione delle funzione "cerca".

## Ma in che modo?

Un primo tentativo potrebbe essere quello di effettuare la ricerca usando dei `T*` (puntatori ad un tipo qualunque), nel modo seguente:

```
template
T* cerca(T* first, T* last, T elem) {
    for ( ; first != last; ++first)
        if (*first == elem)
            return first;
    return last;
}
```

ma sarebbe comunque una scelta limitante. Possiamo immaginare che ci possano essere anche altri tipi di dato, oltre ai puntatori, che possano essere usati in modo analogo. Quindi sostituiamo il tipo `T*` con un ulteriore parametro di template che deve fornire il concetto di iteratore (non necessariamente un puntatore):

```
template
Iter cerca(Iter first, Iter last, T elem) {
    for ( ; first != last; ++first)
        if (*first == elem)
            return first;
    return last;
}
```

Quali sono i requisiti per potere istanziare correttamente il tipo `Iter` nell'algoritmo generico qui sopra?

1. `Iter` deve supportare la copia (passato e restituito per valore).
2. `Iter` deve supportare il confronto binario (`first != last`), per capire se la sequenza è terminata o meno.

3. `Iter` deve supportare il preincremento (`++first`), per avanzare di una posizione nella sequenza.
4. `Iter` deve consentire la dereferenziazione (`*first`), per poter leggere il valore "puntato".
5. Il tipo dei valori puntati da `Iter` deve essere confrontabile con il tipo `T` (usando l'operatore `==`).

Qualunque tipo di dato concreto, che sia o meno un puntatore, se soddisfa questi requisiti (sintattici e semanticci) allora può essere usato per instanziare il mio algoritmo. Si dice che i template applicano delle regole di tipo "strutturali" (in contrapposizione alle regole "nominali"): non importa l'identità del tipo, importa la sua struttura (ovvero le operazioni che rende disponibili e la loro semantica).

Un altro modo di dire le stesse cose (informale e tecnicamente non completamente appropriato) è quello di dire che i template implementano il "duck typing", ovvero un sistema di tipi basato sul "test dell'anatra":

*"If it walks like a duck and it quacks like a duck, then it must be a duck."*

L'anatra quindi è un concetto astratto: qualunque entità che cammina come un'anatra e fa il verso dell'anatra, è un'anatra.

Essendo specificati usando concetti astratti e non classi concrete, gli algoritmi generici risultano di applicabilità più generale. In particolare, non vi sono algoritmi specifici per un dato contenitore della libreria; piuttosto, ogni contenitore fornisce (attraverso i suoi iteratori) la possibilità di essere visto come una sequenza e gli algoritmi lavorano sulle sequenze.

Vedremo quindi esempi di sequenze (per esempio, in sola lettura, in sola scrittura e in lettura/scrittura), tipicamente rappresentate mediante uno o due iteratori; introdurremo inoltre ulteriori concetti astratti (i **callable** e alcune varianti più specifiche, come i predicati) che consentiranno di astrarre la nozione di "chiamata di funzione", consentendoci di parametrizzare gli algoritmi non solo rispetto alla sequenza, ma anche rispetto alle operazioni da applicare sulla sequenza. L'analisi di questi esempi avrà anche l'utile effetto collaterale di farci prendere confidenza con alcuni degli algoritmi generici della libreria standard.

[Torna all'indice](#)

---

## Contenitori associativi

I contenitori associativi sono contenitori che organizzano gli elementi al loro interno in modo da facilitarne la ricerca in base al valore di una "chiave". Abbiamo i seguenti contenitori:

```
std::set  
std::multiset  
std::map  
std::multimap  
std::unordered_set  
std::unordered_multiset  
std::unordered_map  
std::unordered_multimap
```

Queste 8 tipologie di contenitori si ottengono combinando (in tutti i modi possibili) tre distinte proprietà:

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

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

2. La possibilità o meno di memorizzare nel contenitore più elementi con lo stesso valore per la chiave.

Nel caso sia possibile memorizzare più elementi con lo stesso valore per la chiave, abbiamo le versioni "multi" dei contenitori (multinsiemi, multimappe, eccetera).

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  $O(\log(n))$ , al più logaritmico nel numero  $n$  di elementi contenuti (l'implementazione è tipicamente basata su una qualche forma di albero di ricerca bilanciato).

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.

Eventualmente facendo più confronti fino a raggiungere l'elemento cercato o stabilire che non è presente.

[Torna all'indice](#)

---

## Gli adattatori (per contenitori della STL)

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 sono `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 è `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 è stabilita dal criterio di confronto `Cmp`. In questo caso, il contenitore `C` usato per default è uno `std::vector<T>`.

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

[Torna all'indice](#)

# Deduzione automatica dei tipi

---

## Template type deduction

La **template type deduction** (deduzione dei tipi per i parametri template) è il processo messo in atto dal compilatore per semplificare l'istanziazione (implicita o esplicita) e la specializzazione (esplicita) dei template di funzione. Questa forma di deduzione è utile per il programmatore in quanto consente di evitare la scrittura (noiosa, ripetitiva e soggetta a sviste) della lista degli argomenti da associare ai parametri del template di funzione.

Il processo di deduzione è intuitivo e comodo da usare, ma in alcuni casi può riservare sorprese. Per semplificare al massimo, supponiamo di avere la seguente dichiarazione di funzione templatica:

```
template
void f(PT param);
```

nella quale **TT** è il nome del parametro del template di funzione, mentre **PT** indica una espressione sintattica che denota il tipo del parametro **param** della funzione. Il caso che ci interessa, naturalmente, è quello in cui **PT** nomina il parametro templatico **TT**.

Il compilatore, di fronte alla chiamata di funzione **f(expr)**, usa il tipo di **expr(te)** per dedurre:

- un tipo specifico **tt** per **TT**
- un tipo specifico **pt** per **PT**

causando l'istanziazione del template di funzione e ottenendo la funzione

```
void f(pt param);
```

Nota: i tipi dedotti **tt** e **pt** sono correlati, ma spesso non sono identici.

Il processo di deduzione distingue tre casi:

1. **PT** è sintatticamente uguale a **TT&&** (cioè, **PT** è una "universal reference", secondo la terminologia di Meyers).
2. **PT** è un tipo puntatore o riferimento (ma NON una universal reference).
3. **PT** non è né un puntatore né un riferimento.

[Torna all'indice](#)

---

### TT&& (universal reference)

Si ha una **universal reference** quando abbiamo l'applicazione di **&&** al nome di un parametro typename del mio template di funzione, senza nessun altro modificatore. Per esempio, se **TT** è il parametro typename:

```
TT&&           // è una universal reference
const TT&&     // NON è una universal reference (è un rvalue reference)
std::vector&&  // NON è una universal reference (è un rvalue reference)
```

Il nome "universal reference" indica il fatto che, sebbene venga usata la sintassi per i riferimenti a rvalue, può essere dedotto per **PT** un riferimento a rvalue oppure a lvalue, a seconda del tipo **te** di **expr**.

Negli esempi si assume:

```
int i = 0;
const int ci = 1;
```

Esempio 1.1 (rvalue):

```
f(5);           // te = int, deduco pt = int&&, tt = int
f(std::move(i)); // te = int&&, deduco pt = int&&, tt = int
```

Esempio 1.2 (lvalue):

```
f(i);           // te = int&, deduco pt = int&, tt = int&
f(ci);          // te = const int&, deduco pt = const int&, tt = const int&
```

## PT puntatore o riferimento

È diverso da TT&&. Fondamentalmente, si effettua un pattern matching tra il tipo te e PT, ottenendo i tipi tt e pt di conseguenza:

Esempio 2.1:

```
template
void f(TT* param);

f(&i); // te = int*, deduco pt = int*, tt = int
f(&ci); // te = const int*, deduco pt = const int*, tt = int
```

Esempio 2.2:

```
template
void f(const TT* param);

f(&i); // te = int*, deduco pt = const int*, tt = int
f(&ci); // te = const int*, deduco pt = const int*, tt = int
```

Il caso dei riferimenti è analogo:

Esempio 2.3:

```
template
void f(TT& param);

f(i); // te = int&, deduco pt = int&, tt = int
f(ci); // te = const int&, deduco pt = const int&, tt = const int
```

Esempio 2.4:

```
template
void f(const TT& param);

f(i); // te = int&, deduco pt = const int&, tt = int
f(ci); // te = const int&, deduco pt = const int&, tt = int
```

[Torna all'indice](#)

## PT né puntatore né riferimento

```
template
void f(TT param);
```

Siccome abbiamo un passaggio per valore, argomento e parametro sono due oggetti distinti: eventuali riferimenti e qualificazioni const (a livello esterno, quelli a destra di \*) dell'argomento NON si propagano al parametro.

Esempio 3.1:

```
f(i); // te = int&, deduco pt = int, tt = int
f(ci); // te = const int&, deduco pt = int, tt = int
```

Si noti comunque che eventuali qualificazioni const a livello interno vengono preservate:

Esempio 3.2:

```
const char* const p = "Hello";
f(p); // te = const char* const&, deduco pt = const char*, tt = const char*
```

[Torna all'indice](#)

## Auto type deduction

A partire dallo standard C++11, nel linguaggio è stata introdotta la possibilità di definire le variabili usando la parola chiave auto, senza specificarne esplicitamente il tipo, lasciando al compilatore il compito di dedurlo a partire dall'espressione usata per inizializzare la variabile.

Esempio:

```
auto i = 5;           // `i` ha tipo int
const auto d = 5.3;   // `d` ha tipo const double
auto ii = i * 2.0;    // `ii` ha tipo double
const auto p = "Hello"; // `p` ha tipo const char* const
```

La auto type deduction segue (in larga misura) le stesse regole elencate sopra per la template type deduction.

In pratica, quando si osserva una definizione di variabile come

```
auto& ri = ci;
```

1. la parola chiave auto svolge il ruolo del parametro template TT;
2. la sintassi auto& corrisponde a PT;
3. l'inizializzatore ci (di tipo const int&) corrisponde all'espressione expr.

L'esempio qui mostrato corrisponde quindi al caso 2 della deduzione di parametri template (PT è un riferimento a lvalue, non universal). Per auto si deduce il tipo const int e quindi per ri si deduce il tipo const int& (si veda il secondo caso dell'Esempio 2.3).

La forma sintattica auto&& indica una universal reference, che potrebbe dedurre sia un rvalue o un lvalue reference a seconda del tipo dell'inizializzatore.

**La auto template deduction differisce però dalla template type deduction** quando l'inizializzatore è indicato mediante la sintassi che prevede le parentesi graffe, come nell'esempio:

```
auto i = { 1 };
```

In questo caso speciale, che non approfondiremo, l'argomento si considera di tipo std::initializer\_list<T>.

Alcune linee guida di programmazione suggeriscono di usare auto quasi sempre per inizializzare le variabili; nell'acronimo AAA (Almost Always Auto), la prima A (Almost) indica appunto la presenza di eccezioni alla linea guida (quelle viste sopra per gli inizializzatori con parentesi graffe).

[Torna all'indice](#)

# Iteratori

---

## Il concetto di iteratore

È un concetto astratto. Molti algoritmi 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 possono classificare in 5 categorie distinte (che corrispondono, tecnicamente, a 5 concetti correlati ma distinti), che si differenziano per le operazioni supportate e per le corrispondenti garanzie fornite all'utente. Le categorie sono:

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

[Torna all'indice](#)

---

## Iteratori di input

Consentono di effettuare le seguenti operazioni:

- `++iter`: avanzamento di una posizione nella sequenza.
- `iter++`: avanzamento postfisso (NON usarlo: preferire la forma prefissa).
- `*iter`: accesso (in sola lettura) all'elemento corrente.
- `iter->m`: equivalente a `(*iter).m` dove si assume che l'elemento abbia tipo classe e che `m` sia un membro della classe.
- `iter1 == iter2`: confronto (per uguaglianza) tra iteratori: tipicamente usato per verificare se siamo giunti al termine di una sequenza.
- `iter1 != iter2`: confronto per disuguaglianza.

Un esempio di iteratore di input è dato dagli iteratori definiti sugli stream di input `std::istream`, attraverso i quali è possibile leggere i valori presenti sullo stream:

```
#include
#include

int main() {

    // uso di iteratori per leggere numeri double da std::cin
    // inizio della (pseudo) sequenza
    std::istream_iterator i(std::cin);
    // fine della (pseudo) sequenza
    std::istream_iterator iend;

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

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. Per esempio:

```
std::istream_iterator i(std::cin); // inizio della (pseudo) sequenza
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 *invalido*
std::cout << *j; // errore: comportamento NON definito

```

In linguaggio informale, si dice che gli iteratori di input potrebbero essere "*one shot*"; analogamente si dice che potrebbero "*non essere riavvolgibili*" (cioè non consentono di scorrere più volte la stessa sequenza). Intuitivamente, l'operazione di avanzamento *consuma* l'input letto precedentemente (quindi, se lo si volesse rileggere, occorre averlo adeguatamente salvato da qualche altra parte).

[Torna all'indice](#)

---

## Iteratori forward

Consentono di effettuare tutte le operazioni supportate dagli iteratori di input. Inoltre, l'operazione di avanzamento effettuata su un iteratore forward *NON* invalida eventuali altri iteratori che puntano ad elementi precedenti nella sequenza (cioè, gli iteratori forward sono riavvolgibili e consentono di scorrere più volte la stessa sequenza).

Infine, se il tipo dell'elemento indirizzato è modificabile, un iteratore forward può essere usato anche per scrivere (non solo per leggere).

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

```

#include
#include

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

    // Modifica gli elementi della lista
    // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
    // dell'iteratore usato, che sarebbe std::forward_list::iterator
    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 tipo
    // dell'iteratore usato, che sarebbe std::forward_list::const_iterator
    for (auto i = lista.cbegin(); i != lista.cend(); ++i)
        std::cout << *i << std::endl;
}

```

[Torna all'indice](#)

---

## Iteratori bidirezionali

Consentono di effettuare tutte le operazioni supportate dagli iteratori forward (e quindi anche tutte quelle degli iteratori di input). Inoltre, consentono di spostarsi all'indietro sulla sequenza, usando gli operatori di decremento:

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

Esempi di iteratori bidirezionali sono quelli resi disponibili dal contenitore `std::list`. Altri esempi sono gli iteratori resi disponibili dai contenitori associativi (`std::set`, `std::map`, ecc..).

```

#include
#include

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

    // Modifica gli elementi della lista
    // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
    // dell'iteratore usato, che sarebbe std::list::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 tipo
// dell'iteratore usato, che sarebbe std::list::const_iterator
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 tipo
// dell'iteratore usato, che sarebbe std::list::const_reverse_iterator
for (auto i = lista.crbegin(); i != lista.crend(); ++i)
    std::cout << *i << std::endl;
}

```

[Torna all'indice](#)

---

## Iteratori random access

Consentono di effettuare tutte le operazioni supportate dagli iteratori bidirezionali (e quindi anche quelle dei forward e degli input iterator).

Sono inoltre supportate le seguenti operazioni (n è un valore intero):

- iter += n: sposta iter di n posizioni (in avanti se n è positivo, 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, ovvero il numero di elementi che dividono le due posizioni (i due iteratori devono essere definiti sulla stessa sequenza).
- iter1 < iter2: restituisce true se iter1 occorre prima di iter2 nella sequenza (che deve essere la stessa).
- iter1 > iter2: analoghi
- iter1 <= iter2
- iter1 >= iter2

Esempi di iteratori random access sono i puntatori (per esempio sugli *array built-in*) e gli iteratori forniti da std::vector, std::deque, std::array, std::string, std::bitset, ...

```

#include
#include

int main() {
    std::vector 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;
}

```

[Torna all'indice](#)

---

## Iteratori di output

Gli iteratori di output sono iteratori che permettono solamente di scrivere gli elementi di una sequenza: l'operazione di scrittura deve essere fatta una volta sola, dopodiché è necessario incrementare l'iteratore (intuitivamente, per riposizionare correttamente l'iteratore, preparandosi per la scrittura successiva).

Le uniche operazioni consentite sono quindi le seguenti:

- `++iter`: avanzamento di una posizione nella sequenza
- `iter++`: avanzamento postfisso (NON usarlo: preferire la forma prefissa)
- `*iter`: accesso (in sola scrittura) all'elemento corrente

Si noti che NON 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.

Un esempio di iteratore di output è dato dagli iteratori definiti sugli stream di output `std::ostream`, attraverso i quali è possibile scrivere valori di un determinato tipo sullo stream.

```
#include <iostream>
#include <cmath>

int main() {
    std::ostream_iterator<double> out(std::cout, "\n"); // posizione iniziale
    // Nota: non esiste una "posizione finale"
    // Nota: il secondo argomento del costruttore serve da separatore;
    // 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* scrittura
    }
}
```

Si noti che, quando il tipo degli oggetti "puntati" è accessibile in scrittura, gli iteratori forward, bidirezionali e random access soddisfano i requisiti degli iteratori di output e quindi possono essere usati ovunque sia necessario fornire un iteratore di output.

[Torna all'indice](#)

---

## Il template di classe iterator\_traits

```
std::iterator_traits
```

Come si è sottolineato, alcune categorie di iteratori implementano un sovrainsieme delle operazioni e garanzie fornite da altre categorie: ciò significa che ogni volta che, nella documentazione di una funzione generica, si afferma che il parametro di tipo `Iter` è richiesto essere (per esempio) un *iteratore forward*, l'utente può istanziare correttamente quel parametro di template usando un qualunque iteratore concreto delle categorie forward, bidirezionale e random access.

L'utente commetterebbe però un errore se istanziasse il parametro `Iter` con uno `std::istream_iterator`, perché questi sono (solo) iteratori di input.

Quando abbiamo visto le interfacce dei contenitori standard, abbiamo notato come essi forniscano un certo numero di alias di tipo che consentono di dare nomi "*canonici*" ad alcuni tipi utili nella definizione e uso dell'interfaccia stessa (`size_type`, `value_type`, `iterator`, ecc.).

La necessità di usare nomi canonici è avvertita anche quando si scrivono algoritmi generici che sfruttano il concetto di iteratore, ragione per cui un iteratore implementato come classe dovrebbe fornire i seguenti type alias:

- `value_type`: tipo ottenuto dereferenziando l'iteratore.
- `reference`: tipo riferimento (al `value_type`).
- `pointer`: tipo puntatore (al `value_type`).
- `difference_type`: tipo intero con segno (per le "distanze" tra iteratori).
- `iterator_category`: un tipo "tag" (marcatore), che indica la categoria dell'iteratore.

[Torna all'indice](#)

## Osservazioni

Non sono forniti i `const_reference` e `const_pointer`, perché è l'iteratore che decide se il `value_type` è o meno in sola lettura; per esempio, se da un vettore `vi` di tipo `const std::vector<int>&` estraggo un iteratore usando il metodo `begin()`, otterò un `std::vector<int>::const_iterator` il cui alias `reference` è `const int&` e il cui alias `pointer` è `const int*`.

La `iterator_category` è un "*tag type*" (ovvero un tipo che può assumere un solo valore, il cui unico significato è dato dall'identità del tipo stesso). I tipi tag per le categorie di iteratori sono definiti nella libreria standard in questo modo:

```
struct output_iterator_tag { };

struct input_iterator_tag { };

struct forward_iterator_tag
: public input_iterator_tag { };

struct bidirectional_iterator_tag
: public forward_iterator_tag { };

struct random_access_iterator_tag
: public bidirectional_iterator_tag { };
```

Le relazioni di ereditarietà dicono, per esempio, che un `bidirectional_iterator_tag` può essere convertito implicitamente (tramite up-cast) ad un `forward_iterator_tag` o ad un `input_iterator_tag`, ma *NON* può essere convertito ad un `random_access_iterator_tag`.

Queste conversioni codificano le relazioni esistenti tra le categorie di iteratori, dicendo per esempio che un `bidirectional` è accettabile quando viene richiesto un `forward`, ma non vale il viceversa. Questi tag types possono quindi essere usati per codificare versioni alternative di un algoritmo generico scelte in base alla categoria dell'iteratore (come esempi concreti, vedere le funzioni generiche `std::advance` e `std::distance`).

Abbiamo detto che in linea di principio ogni iteratore concreto dovrebbe fornire gli alias di tipo descritti sopra. Ma come farlo? Non possiamo adottare banalmente la tecnica usata per i contenitori standard, perché tra i nostri iteratori ci sono anche tipi che *NON* sono classi (i puntatori) e che quindi *NON* consentono di essere interrogati mediante la sintassi che usa l'operatore di scope:

```
Iter::value_type // con un typename prefisso, se necessario
```

Il problema si risolve usando il template di classe `std::iterator_traits`: invece di interrogare direttamente il tipo iteratore, si interroga la classe traits ottenuta istanziando il template con quel tipo iteratore. Per esempio, se vogliamo conoscere il value type di `Iter`, scriviamo

```
std::iterator_traits::value_type
```

L'uso di `iterator_traits` è solo uno degli esempi di uso di classi "*traits*", ovvero tipi di dato che hanno lo scopo di fornire qualche informazione (traits, ovvero le caratteristiche) di altri tipi di dato.

In particolare, le classi *traits* consentono di effettuare queste analisi di "introspezione" anche sui tipi built-in (che non forniscono direttamente meccanismi per l'introspezione).

Altri esempi, già intravisti, di classi traits sono:

- Template di classe `std::numeric_limits<T>`: consente di interrogare tipi numerici per ottenere informazioni quali i valori minimi e massimi rappresentabili, la signedness, il fatto di supportare o meno calcoli esatti, ecc.
- Template di classe `std::char_traits<T>`: consente di interrogare i tipi carattere per ottenere accesso, per esempio, alle funzioni di confronto da usare per l'ordinamento lessicografico.

Dal punto di vista dell'utilizzo, il template `std::iterator_traits` è banale. E' però interessante vederne l'implementazione, perché fornisce un esempio semplice di specializzazione *parziale* di template di classe.

Intuitivamente, gli `iterator_traits` devono distinguere i tipi puntatore dagli altri iteratori (di tipo definito dall'utente). Nel secondo caso, che corrisponde al template non specializzato, si "*delega*" al tipo `Iter` definito dall'utente il compito di

fare il lavoro di introspezione:

```
template
struct iterator_traits {
    typedef typename Iter::iterator_category iterator_category;
    typedef typename Iter::value_type           value_type;
    typedef typename Iter::difference_type     difference_type;
    typedef typename Iter::pointer             pointer;
    typedef typename Iter::reference          reference;
};
```

Quando invece il tipo `Iter` è un puntatore, il template di classe si attiva per restituire all'utente le informazioni che il tipo built-in non sarebbe in grado di fornire. Vengono quindi fornite due specializzazioni parziali (*non-const* e *const*) del template di classe:

```
template
struct iterator_traits {
    typedef random_access_iterator_tag iterator_category;
    typedef T                         value_type;
    typedef ptrdiff_t                 difference_type;
    typedef T*                        pointer;
    typedef T&                       reference;
};

template
struct iterator_traits {
    typedef random_access_iterator_tag iterator_category;
    typedef T                         value_type;
    typedef ptrdiff_t                 difference_type;
    typedef const T*                 pointer;
    typedef const T&                reference;
};
```

[Torna all'indice](#)

# Obiettivo

Lettura stringe da input e contare le occorrenze.

Per indicare la fine dell'inserimento dell'input utilizziamo **ctrl + D**

---

```
#include
#include
#include
#include

using WordFreq = std::map;

// criterio di confronto
struct Greater{
    bool operator()(unsigned long x, unsigned long y) const {
        return x > y;
    }
};

// Conta le stringhe di lunghezza n
struct Shorter{
    bool operator()(const std::string& x, const std::string& y) const {
        return x.size() < y.size();
    }
};

using FreqWord = std::multimap;
// un'alternativa a greater è std::greater

int main(){
    std::cout << "`ctrl + d` per terminare l'inserimento \n";
    std::cout << "Inserire input:\n";

    WordFreq wf;

    std::istream_iterator i(std::cin);
    std::istream_iterator iend;

    // Versione Zaffanella
    for( ; i != iend; ++i){
        const auto& s = *i;
        auto iter = wf.find(s);

        if(iter == wf.end()) // se non trovo niente
            wf.insert(std::make_pair(s, 1));
        else // se trovo qualcosa
            ++iter->second; // oppure ++(*iter).second;
    }

    /** Versione del libro (Stroustrup), che però è poco leggibile
    for( ; i != iend; ++i)
        ++wf[*i];
    */

    // Serve per stampare le occorrenze in ordine decrescente
    FreqWord fw;

    for(const auto& p : wf){
        fw.insert(std::make_pair(p.second, p.first));
    }

    // for(auto iter = wf.begin(); iter != wf.end(); ++iter)
    for(const auto& p : fw){
        std::cout << "La stringa " << p.first
            << " occorre numero " << p.second
            << " volte \n";
    }

    return 0;
}
```

# Callable

---

## Il concetto callable

Molti algoritmi generici resi disponibili dalla libreria standard sono forniti in due differenti versioni, la seconda delle quali è parametrizzata rispetto a una "policy".

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
FwdIter adjacent_find(FwdIter first, FwdIter last);

template
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==`.

Si noti che 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

```
if (pred(*first, *next)) //...
```

dove `first` e `next` sono due iteratori dello stesso tipo.

Considerando un caso un po' meno specifico, ci chiediamo quali siano i tipi di dato `Fun` che consentono (ai propri valori `fun`) di essere intuitivamente utilizzati come nomi di funzione in una chiamata:

```
fun(arg1, ..., argN);
```

L'insieme di questi tipi di dato forma il concetto "*callable*" (i tipi "chiamabili", cioè "invocabili" come le funzioni):

- puntatori a funzione
- oggetti funzione
- expressioni lambda (dal C++11)

[Torna all'indice](#)

---

## I puntatori a funzione

Nei pochi esempi concreti che abbiamo visto fino ad ora, abbiamo sempre istanziato i parametri "callable" usando un opportuno puntatore 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`).

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.

[Torna all'indice](#)

---

## Gli "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 {  
        return i % 2 == 0;  
    }  
};  
  
int foo() {  
    Pari pari;  
    if (pari(12345)) ...  
    ...  
}
```

L'oggetto `pari` (di tipo `struct Pari`) non è una funzione ma, essendo fornito di un metodo `operator()`, può essere invocato come una funzione.

Si noti che 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 dati membri) e quindi non sono modificati dalle invocazioni dei loro `operator()`.

[Torna all'indice](#)

---

## Osservazioni

A prima vista, gli oggetti funzione potrebbero sembrare un modo complicato di risolvere un problema semplice: perché definire una classe con un metodo `operator()` quando posso, più semplicemente, passare direttamente il nome di una semplice funzione?

Da un punto di vista tecnico, gli oggetti funzione possono essere usati per ottenere un vantaggio in termini di efficienza rispetto alle normali funzioni: in particolare, l'uso degli oggetti funzione fornisce al compilatore più opportunità per l'ottimizzazione del codice.

Si consideri un programma che lavori su un `vector` di interi e istanzia più volte la funzione generica `std::find_if` per effettuare ricerche nel `vector` usando criteri di ricerca (cioè, predicati unari) diversi.

Consideriamo i seguenti predicati espressi mediante funzioni:

```
bool pari(int i);  
bool dispari(int i);  
bool positivo(int i);  
bool negativo(int i);  
bool maggiore_di_1000(int i);  
bool numero_primo(int i);
```

Queste sei funzioni hanno tutte lo stesso tipo `bool(int)`, ovvero funzione che prende un argomento `int` per valore e restituisce un `bool`. Di conseguenza, quando si istanzia l'algoritmo `std::find_if` sul vettore usando i sei predicati, si

ottiene ogni volta la stessa identica istanza del template di funzione.

Se, per comodità, usiamo i seguenti alias di tipo

```
using Iter = std::vector::iterator  
using Ptr = bool (*) (int);
```

la specifica istanza ottenuta sarà la seguente:

```
Iter std::find_if(Iter first, Iter last, Ptr pred);
```

Il codice generato, quindi, è unico e deve gestire correttamente tutte le sei possibili invocazioni: di conseguenza, la chiamata al predicato è implementata come chiamata di funzione (attraverso il puntatore a funzione).

Supponiamo ora che i sei predicati siano stati invece implementati mediante oggetti funzione, ovvero definendo sei classi **Pari**, **Dispari**, **Positivo**, ecc...

Nota: l'uso dell'iniziale maiuscola è solo una convenzione, utile per evitare di fare confusione.

In questo caso, quando si istanzia l'algoritmo `std::find_if` sul vettore usando i sei oggetti funzione, siccome i tipi degli oggetti funzione sono distinti si otterranno sei diverse istanze del template di funzione:

```
Iter std::find_if(Iter first, Iter last, Pari pred);  
Iter std::find_if(Iter first, Iter last, Dispari pred);  
Iter std::find_if(Iter first, Iter last, Positivo pred);  
...
```

Quando genera il codice per una delle sei istanze, il compilatore vede l'invocazione di uno solo dei sei metodi `operator()` e quindi può ottimizzare il codice per quella specifica invocazione (per esempio, facendo l'espansione in linea della chiamata). Si ottiene quindi un codice eseguibile più grande (sei istanze invece di una sola), ma meglio ottimizzabile e quindi potenzialmente più efficiente.

[Torna all'indice](#)

---

## Le espressioni lambda

Capita frequentemente che una determinata funzione (o un oggetto funzione) debba essere fornita come callable ad una invocazione di un algoritmo generico. In alternativa, che una funzione (o oggetto funzione) debba essere definita a fronte di un unico punto del codice che la invoca.

In questi casi, fornire la definizione della funzione (o della classe che implementa un oggetto funzione equivalente) presenta alcuni svantaggi:

- occorre inventare un nome appropriato;
- occorre fornire la definizione in un punto diverso del codice rispetto all'unico punto di uso, potenzialmente distante.

Le espressioni lambda (dette anche funzioni lambda) forniscono una comoda sintassi abbreviata per potere definire un oggetto funzione "anonimo" e immediatamente utilizzabile.

Nota: le espressioni lambda sono state introdotte con lo standard C++11 e sono state oggetto di estensioni negli standard C++14 e C++17.

Esempio: istanziazione di `std::find_if` con una lambda expression che implementa il predicato `pari` sul tipo T.

```
void foo(const std::vector& v) {  
    auto iter = std::find_if(v.begin(), v.end(),  
                           [] (const long& i) {  
                               return i % 2 == 0;  
                           });  
    // ... usa iter  
}
```

L'espressione lambda è 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
```

In questo esempio, la lista delle catture è vuota; inoltre, il tipo di ritorno è omesso, in quanto viene dedotto dall'istruzione di return contenuta nel corpo della funzione. Volendo (ma di solito non si fa), è possibile specificarlo con la sintassi del "*trailing return type*", che usa l'operatore freccia:

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

Questo uso della espressione lambda all'interno della invocazione della `std::find_if` corrisponde intuitivamente alle seguenti operazioni:

1. Definizione di una classe "anonima" per oggetti funzione, cioè una classe dotata di un nome univoco scelto dal sistema.
2. Definizione all'interno della classe di un metodo `operator()` che ha i parametri, il corpo e il tipo di ritorno specificati (o dedotti) dalla lambda expression.
3. Creazione di un oggetto funzione "*anonimo*", avente il tipo della classe anonima suddetta, da passare alla `std::find_if`.

In pratica, è 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 può essere usata quando l'espressione lambda deve potere accedere a variabili locali visibili nel punto in cui viene creata (che è diverso dal punto in cui verrà invocata). Supponiamo per esempio di volere trovare il primo valore della sequenza che sia maggiore del parametro "soglia":

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

In questo caso, la definizione della lambda è 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 virgolette. 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隐式的 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 nel corpo per valore.
- `[&]` --> cattura (implicitamente) ogni variabile locale usata nel corpo per riferimento.

- `[=, &pippo, &pluto] --> cattura (implicitamente) per valore, tranne pippo e pluto che sono catturate per riferimento.`
- `[&, =pippo, =pluto] --> cattura (implicitamente) per riferimento, tranne pippo e pluto che sono catturate per valore.`

Il consiglio è di effettuare sempre catture esplicite, per maggiore leggibilità del codice.

Si noti che il metodo `operator()` definito nella classe è qualificato `const`; di conseguenza, le variabili catturate possono essere accedute in sola lettura. Se si vuole consentirne la modifica, occorre aggiungere alla lambda il modificatore `mutable`.

[Torna all'indice](#)

---

## Esempio

Modifichiamo la lambda dell'esempio per tenere traccia del numero di sue invocazioni.

```
void foo(const std::vector& v) {
    long num_chiamate = 0;
    auto iter = std::find_if(v.begin(), v.end(),
        [&num_chiamate](const long& i) mutable {
            ++num_chiamate;
            return i % 2 == 0;
        });
    std::cout << "Funzione lambda invocata " << num_chiamate << " volte\n";
}
```

Il modificatore `mutable` viene associato a tutti i dati membro catturati: esso consente anche ad un metodo marcato `const` di accedere in scrittura al dato membro.

Si noti che abbiamo catturato per riferimento, perché vogliamo che venga modificata proprio la variabile locale della funzione `foo` (non una sua copia).

La classe che viene generata implicitamente è quindi simile alla seguente:

```
struct Nome_Univoco {
    mutable long& num_chiamate;
    Nome_Univoco(long& nc) : num_chiamate(nc) { }
    bool operator()(const long& i) const {
        ++num_chiamate;
        return i % 2 == 0;
    }
};
```

Nel caso vengano effettuate catture per riferimento, occorre prestare attenzione a NON usare la funzione lambda dopo che il tempo di vita della variabile catturata è terminato (si incorrerebbe in undefined behavior).

**NOTA:** come detto, 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& v,
                  const std::list& l,
                  unsigned max_size) {
    auto corta = [max_size](const std::string& s) {
        return s.size() <= max_size;
    };
    std::ostream_iterator out(std::cout, "\n");
    out = std::copy_if(v.begin(), v.end(), out, corta);
    out = std::copy_if(l.begin(), l.end(), out, corta);
}
```

[Torna all'indice](#)

# Overloading e template di funzione

Quando sono state introdotte le regole di risoluzione dell'overloading si era accennato al fatto che esistono regole speciali nel caso in cui alcune funzioni siano state ottenute come instanziazione o specializzazione di template di funzione.

Si ricorda che, in senso tecnico, i template di funzione NON sono funzioni. Vale forse la pena ripeterlo: sono degli schemi per generare, mediante istanziazione, un numero potenzialmente illimitato di funzioni vere e proprie. Ad esempio, posso prendere l'indirizzo di una istanza del template, ma non esiste un indirizzo del template.

Quindi quando si dice (informalmente) che un template di funzione è nell'insieme delle candidate nella risoluzione dell'overloading, in realtà si intende che una sua specifica istanza (o specializzazione totale) è candidata.

---

## Ordinamento parziale dei template di funzione

I template di funzione dotati dello stesso nome e visibili nello stesso scope (i.e., possono andare in overloading) sono ordinati parzialmente rispetto ad una relazione di "specificità". Anche in questo caso, la regola precisa è complicata e ne forniamo una versione semplificata.

Intuitivamente, denotiamo con  $\text{Istanze}(X)$  l'insieme di tutte le possibili istanze del template  $X$ . Allora si dice che il template di funzione  $X$  è più specifico del template di funzione  $Y$  se  $\text{Istanze}(X)$  è un sottoinsieme proprio di  $\text{Istanze}(Y)$ .

---

## Regola speciale per i template di funzione

Nella risoluzione dell'overloading, le istanze dei template più 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 è possibile effettuare la deduzione dei parametri del template. Durante il processo di deduzione *NON* si possono applicare promozioni, conversioni standard e conversioni definite dall'utente (in pratica, si possono applicare solo le corrispondenze esatte).

Nota: se il processo di deduzione ha successo, allora l'istanza del template oltre ad essere candidata è anche utilizzabile.

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

# Classi dinamiche

---

## 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 ... */  
};
```

Come già sappiamo, 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;
```

Se la derivazione fosse non 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.

Intuitivamente, l'esistenza di questa conversione indica che è possibile utilizzare un oggetto Derived (tipo concreto) come se fosse un oggetto della classe Base (tipo astratto), ignorando eventuali caratteristiche specifiche della classe Derived per concentrarsi sulle caratteristiche che questa classe ha in comune con (eredita da) la classe Base.

Si dice che la classe Derived è in relazione "IS-A" con la classe Base, cioè è una particolare concretizzazione della classe Base, e quindi deve potere essere utilizzato, dall'utente, come se fosse un oggetto di tipo Base.

In altre parole, in questo contesto l'utente vuole lavorare con oggetti di tipo Base, ignorando eventuali differenze tra le varie concretizzazioni possibili.

[Torna all'indice](#)

---

## Esempio 1

Dalla classe base Docente possiamo derivare tante classi in relazione IS-A, come Professore\_Ordinario, Professore\_Associato, Ricercatore, Professore\_a\_Contratto, ecc. Ognuna di queste classi potrebbe avere caratteristiche (dati o metodi) specifici che la differenziano dalle altre. Un utente che NON sia interessato a queste peculiarità può astrarre da esse, vedendo tutti gli oggetti concreti come istanze di Docente e usando solo l'interfaccia messa a disposizione dalla classe base Docente.

**NOTA:** in contesti diversi la derivazione potrebbe essere utilizzata con altri scopi. Per esempio, a volte si usa (secondo alcuni, a sproposito) l'ereditarietà per codificare la relazione "HAS-A": la classe Derived ha un sotto-oggetto di tipo Base, ovvero lo "possiede" e quindi lo può usare.

Per esempio: un Automezzo ha un Motore e, siccome più automezzi di tipo diverso possono usare lo stesso tipo di motore, si potrebbe decidere di usare Motore come classe base comune ai vari automezzi concreti.

La differenza sostanziale, rispetto al caso precedente, è data dal fatto che l'utente di queste classi, probabilmente, è interessato ad usare gli automezzi concreti (e non i motori in essi contenuti, che potrebbero essere visti come dei dettagli implementativi): quindi, l'utente NON è interessato alla possibilità di convertire un automezzo concreto in un Motore, per cui l'uso di ereditarietà pubblica è inappropriato.

Le alternative sono:

1. uso di ereditarietà privata:

```
class Utilitaria : private Motore { /* ... */ };
```

## 2. uso del contenimento:

```
class Utilitaria { Motore motore; /* ... */ };
```

Tra le due opzioni, dovrebbe essere preferita la seconda, in quanto più intuitiva da usare; la seconda opzione, inoltre, è facilmente estendibile al caso in cui l'automezzo debba contenere più di un solo sotto-oggetto di un determinato tipo (esempio: auto ibride con più motori).

Nel seguito, ci concentreremo sul caso in cui l'utente sia intenzionato a stabilire relazioni di tipo "IS-A", usando quindi l'ereditarietà pubblica e sfruttando le conversioni implicite allo scopo di lavorare con la classe base, astraendo dai dettagli implementativi delle classi derivate.

[Torna all'indice](#)

---

## Metodi virtuali e classi dinamiche

**Nota bene:** nel seguito sono mostrati spezzoni di codice, incompleti; il loro unico scopo è quello di consentire al lettore di "immaginare" un contesto concreto, ma semplificato al massimo, nel quale applicare le nozioni di cui si sta trattando.

Consideriamo il seguente esempio

```
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 chiamante invocherà la funzione `stampa_tutti` passando un puntatore ad una stampante concreta (una specifica istanza di `FilePrinter` o `NetworkPrinter`), sfruttando l'up-cast consentito dalla relazione "IS-A". Quando esamina la chiamata al metodo `print`, il compilatore si troverà a fare la risoluzione dell'overloading conoscendo solo il tipo statico di `printer` (puntatore alla classe base `Printer`), senza avere conoscenza di quello che è il vero tipo dinamico (puntatore ad una delle specifiche classi derivate dalla classe base): di conseguenza, effettuerà la ricerca delle candidate nella classe `Printer` e troverà solo il metodo `Printer::print`, che verrà scelto come migliore funzione utilizzabile.

In realtà, però, l'utente vorrebbe che fosse invocato il metodo specifico della stampante concreta passata alla funzione, che potrebbe dovere fare operazioni diverse a seconda della classe di appartenenza (per esempio, una `NetworkPrinter` potrebbe tenere traccia del numero di pagine stampate dai vari utenti). Intuitivamente, ogni classe concreta "ridefinisce" il metodo `print` per fargli fare la cosa corretta per il contesto specifico: questa ridefinizione del metodo dovrebbe prevalere (override) rispetto a quella della classe. Serve quindi un meccanismo tecnico che consenta di interrogare (a tempo di esecuzione) il puntatore, allo scopo di capire quel è il suo tipo dinamico e quindi "ridirezionare" la chiamata del metodo `print` alla classe concreta corretta: questo meccanismo tecnico effettua la cosiddetta "risoluzione dell'overriding" e, nel caso del C++, 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);
};
```

Una classe che contenga almeno un metodo virtuale viene detta **classe dinamica**, in quanto per gli oggetti di questa classe vengono messe a disposizione le funzionalità che consentono di implementare la risoluzione dell'overriding e, più in generale, la *RTTI* (Run-Time Type Identification).

A livello implementativo, ad ogni oggetto che è istanza di una classe dinamica viene associato un puntatore (non accessibile direttamente da parte dell'utente) usando il quale il RTS (Run-Time Support) del linguaggio può raggiungere le informazioni di tipo della classe. L'esistenza di questo puntatore si può notare se si confrontano, usando l'operatore `sizeof`, oggetti di classi dinamiche e statiche (cioè non dinamiche):

```
#include <iostream>

class Statica {
    ~Statica() {}
};

class Dinamica {
    virtual ~Dinamica() {}
};

int main() {
    std::cout << "sizeof(Statica) = " << sizeof(Statica) << std::endl;
    std::cout << "sizeof(Dinamica) = " << sizeof(Dinamica) << std::endl;
}
```

Avendo dichiarato virtuale il metodo

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

una classe derivata da `Printer` che lo ridefinisca (usando lo stesso nome e lo stesso numero e tipo degli argomenti) ne fa l'overriding; si noti che, se il metodo non fosse stato dichiarato `virtual` nella classe base, NON si avrebbe overriding, ma si avrebbe invece hiding. Nella classe derivata non è necessario (ma è consentito) ripetere la parola chiave `virtual`. Se si usa lo standard C++11 o superiore è anche consigliato usare la parola chiave `override`, da usarsi alla fine della dichiarazione, in questo modo:

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

L'uso di "override" è utile perché causa un errore nel caso in cui nella classe base `Printer` non esista un metodo virtuale corrispondente. L'errore è meno frequente di quanto si possa immaginare, perché:

1. potremmo esserci dimenticati di usare la parola chiave `virtual` nella classe base;
2. potremmo avere modificato leggermente il tipo del metodo, cambiando il numero o il tipo dei parametri.

[Torna all'indice](#)

---

## Esempio 2

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

class NetworkPrinter : public Printer {
public:
    // Errore: in Printer::name il parametro implicito this è qualificato const.
    std::string name() override;
    // Errore: il metodo Printer::print non è dichiarato virtual.
    void print(const Doc& doc) override;
};
```

## Metodi virtuali puri e classi astratte

Quando si definisce una classe base come `Printer`, spesso non si ha la possibilità di fornire una implementazione sensata per i metodi virtuali. Intuitivamente, la classe `Printer` fornisce "solo" l'interfaccia del concetto astratto di stampante e di conseguenza non può stampare davvero un documento: l'unico suo scopo è quello di ridirezionare la chiamata ad una delle classi concrete. Invece di fornire una implementazione fittizia (per esempio, una che lancia una eccezione) è preferibile indicare che il metodo virtuale è "puro", usando la sintassi "`= 0`" al termine della sua dichiarazione.

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

Una classe che contenga metodi 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). Si noti che 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 3

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

## 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 4

```

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

class Concreta : public Astratta {
    std::vector 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 solo
    // per fargli stampare qualcosa, così che sia evidente il fatto che
    // 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 concreta.
    delete a; // invoca il distruttore di Astratta (che non è virtual)
}

```

---

## 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).

[Torna all'indice](#)

# Cast

---

## Conversioni esplicite di tipo in C++

Il \$C\$++ fornisce varie sintassi per effettuare il cast (conversione esplicita di tipo) di una espressione, allo scopo di ottenere un valore di un tipo (potenzialmente) diverso:

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.

[Torna all'indice](#)

---

## Classificazione delle motivazioni per l'uso di cast esplicativi

Essendo conversioni esplicative 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:

1. Il cast implementa una conversione di tipo che NON è consentita dalle regole del linguaggio come conversione implicita, in quanto considerata una frequente fonte di errori di programmazione. Il programmatore, richiedendo esplicitamente la conversione con il cast, si assume la responsabilità della sua correttezza. Esempio:

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

2. 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 è effettivamente consentita. Esempio:

```
struct B { /* ... */ };
struct D : public B { /* ... */ };

void foo(B* b_ptr) {
    if (D* d_ptr = dynamic_cast(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 D
    }
}
```

3. Il cast NON è strettamente necessario (in quanto la corrispondente conversione implicita è consentita dal linguaggio), ma il programmatore preferisce comunque la forma esplicita a scopo di documentazione, per tenere una 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 (anche se non lo è dal punto di vista tecnico). Esempio:

```

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

```

4. Un caso speciale di uso (qualcuno potrebbe pure dire "abuso") di un cast esplicito riguarda la conversione di una espressione al tipo void (che non ha valori), che intuitivamente corrisponde ad 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 (questa convenzione è rispettata sia da g++ che da clang++). Esempio:

```

// Il parametro size lo si usa solo nella assert e quindi, quando le
// asservizioni NON sono attivate, il compilatore mi segnalerebbe
// 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(size);
    /* ... codice che non usa size ... */
}

```

In questo caso spesso si usa, per convenzione, un cast stile \$C\$: (void) size. In realtà, questo è l'unico caso in cui l'uso di un cast stile \$C\$ (in \$C\$++) è tollerato: ogni altro uso è considerato (giustamente) cattivo stile.

[Torna all'indice](#)

---

## Tipologie di cast

Descriviamo ora brevemente le diverse tipologie di cast:

- [[#static\_cast]]
- [[#dynamic\_cast]]
- [[#const\_cast]]
- [[#reinterpret\_cast]]
- [[#cast funzionale]]
- [[#cast stile C]]

[Torna all'indice](#)

---

## static\_cast

Probabilmente, è il cast utilizzato più frequentemente. La sintassi

```
static_cast(expr)
```

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

- è legittima la corrispondente conversione implicita (caso banale);

```

double d = 3.14;
int approx = static_cast(d);

```

- è legittima la costruzione diretta di un oggetto di tipo T passando expr come argomento;

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

- si effettua la conversione inversa rispetto ad un 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_cast(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 è void.

[Torna all'indice](#)

## dynamic\_cast

Il dynamic\_cast è 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 è una conversione consentita anche implicitamente e quindi non necessita della RTTI.
- **down-cast**: conversione da classe base a classe derivata; è il caso più frequente di utilizzo del dynamic 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'ereditarietà 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 non 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 la funzione

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

Se la classe B è dinamica (ovvero, se contiene almeno un metodo virtuale) allora è dotata delle informazioni per la *RTTI* e possiamo applicare cast dinamici ai puntatori per sapere se sono di un determinato tipo. Per esempio:

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

Dopo l'esecuzione di questo cast, se il puntatore `b_ptr` punta ad un oggetto di tipo `D1` (incluso eventualmente un oggetto di una classe derivata, anche indirettamente, da `D1`), allora `d1_ptr` avrà assegnato un valore NON nullo. Se invece `b_ptr` non punta ad un oggetto di tipo `D1` (per esempio, punta ad un oggetto di tipo `D2`), allora a `d1_ptr` viene assegnato il puntatore nullo. Di conseguenza, il programmatore può sapere se il cast è andato a buon fine controllando se il puntatore è non nullo:

```
if (d1_ptr != nullptr) {
/* d1_ptr è valido */
}

if (d1_ptr) {
/* equivalente: ho sfruttato la conversione a bool */
}

if (D1* d1_ptr = dynamic_cast(b_ptr)) {
/* equivalente: ho compattato cast e test */
}
```

Il caso di conversione per un riferimento è diverso (e raro), perché non esiste il concetto di riferimento nullo e quindi NON possiamo usare facilmente cast dinamici su riferimento per fare dei test RTTI. Se proviamo ad eseguire questo:

```
D1& d1_ref = dynamic_cast(*b_ptr)
```

se `b_ptr` punta ad un `D1`, il cast va a buon fine e `d1_ref` è inizializzato correttamente; se invece NON punta a `D1`, il cast dinamico fallisce e, non potendo segnalare la cosa con il riferimento nullo, genera una eccezione (di tipo `std::bad_cast`).

[Torna all'indice](#)

---

## 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` (cioè non modificabile) per ottenere un riferimento o puntatore ad un oggetto non qualificato (e quindi modificabile).

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

La funzione ha promesso al chiamante che NON modificherà l'argomento, ma si rimangia la promessa, elimina la qualificazione `const` e poi modifica l'argomento (proprio quello passato dal chiamante, non una copia).

Usando il `const_cast`, quindi, potremmo "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.

Esempio: Una classe mantiene un collezione di elementi ed è fornita di un metodo (etichettato `const`) che stampa gli elementi secondo un dato ordinamento. L'ordinamento è costoso da calcolare e quindi la collezione è mantenuta internamente NON ordinata. Quando però mi viene richiesta una stampa ordinata, potrei decidere di modificare la rappresentazione interna allo scopo di memorizzare la sequenza ordinata (di modo che successive chiamate della routine di stampa siano più efficienti). In questo caso, la routine di stampa potrebbe usare un `const_cast` per modificare la rappresentazione interna (senza però modificare dal punto di vista semantico la collezione).

Nota: alcuni usi di `const_cast` si potrebbero eliminare mediante l'utilizzo del modificatore `mutable` su alcuni dati membro di una classe.

[Torna all'indice](#)

---

## 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).

Non è possibile usare un `reinterpret_cast` per rimuovere la qualificazione `const` (occorre usare il `const_cast`).

Nel caso del `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 più pericolose, in quanto i controlli di correttezza sono lasciati quasi completamente nelle mani del programmatore.

[Torna all'indice](#)

---

## cast funzionale

La sintassi `T(expr)` oppure `T()`, dove `T` è il nome di un tipo, viene spesso indicata come "cast funzionale". Intuitivamente, corrisponde alla costruzione diretta di un oggetto di tipo `T`, usando un 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 built-in (che in senso tecnico non sono dotati di costruttori). Nel caso di un tipo built-in, la forma `T()` produce la cosiddetta zero-initialization.

Esempio:

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

Se `foo` viene istanziata con `T = int` e `U = double`, il test condizionale diventa

```
if (t == int(u))
```

nel quale abbiamo il cast funzionale `int(u)`.

[Torna all'indice](#)

---

## cast stile C

Hanno la sintassi

```
(T) expr
```

Il loro uso è considerato cattivo stile (tranne il caso nominato sopra del 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.

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

[Torna all'indice](#)

# Principi progettazione object oriented

---

## L'acrostico SOLID

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

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

S \$\to\$ SRP (Single Responsibility Principle) O \$\to\$ OCP (Open-Closed Principle) L \$\to\$ LSP (Liskov Substitution Principle) I \$\to\$ ISP (Interface Segregation Principle) D \$\to\$ DIP (Dependency Inversion Principle)

[Torna all'indice](#)

---

## Note importanti

I principi SOLID non sono stati "inventati" o "proposti" contemporaneamente: l'acrostico ha solamente fornito un nome facilmente memorizzabile per concetti o metodologie che sono stati sviluppati in tempi diversi da persone diverse e, abbastanza spesso, con nomi diversi. L'uso sistematico dei TLA (Three Letter Acronyms) per riferirsi ai singoli principi può essere etichettato come una "moda" dei tempi.

L'ordine in cui sono elencati i principi NON corrisponde ad una qualche relazione di priorità tra questi, ma è solo funzionale a creare l'acrostico. Di conseguenza, verranno presentati in un ordine diverso.

Si parla di "principi" e non di "tecniche" o "metodi", perché non sono immediatamente applicabili (e a maggior ragione non è opportuno immaginare che vengano applicati in maniera sistematica e/o automatica).

La loro applicazione richiede uno sforzo per valutare se sia o meno opportuno applicarli nei vari contesti concreti che si presentano, in quanto ogni beneficio (conseguente all'adozione di una modifica di progetto che favorisca uno dei principi) spesso comporta anche un corrispondente costo (in termini di comprensibilità del codice da parte del programmatore e/o in termini di modifiche da applicare a codice esistente).

In linea di massima, il progettista dovrebbe individuare quelle parti del software che:

- sono in contrasto con alcuni di questi principi;
- in futuro potrebbero beneficiare dall'applicazione di una ristrutturazione del codice in linea con i principi;

e quindi applicare le azioni correttive del caso.

La pretesa di applicare i principi sistematicamente, a tutte le porzioni di codice, indipendentemente da ogni valutazione di opportunità, tipicamente porta a progetti oltremodo complicati, che violano altri principi di progettazione. A titolo di esempio, si ricorda il principio KISS (Keep It Simple, Stupid) che suggerisce di evitare ogni tipo di complicazione non strettamente necessaria.

[Torna all'indice](#)

---

## SRP (Single Responsibility Principle)

Si tratta di un principio di validità generale (cioè, non è 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**.

A volte si dice che ogni classe dovrebbe avere un solo "motivo per essere modificata": se esistono più motivi distinti, questo è indice che la classe si assume più responsabilità e quindi dovrebbe essere suddivisa in più componenti, ognuno

dei quali caratterizzato da una singola responsabilità.

Il rispetto del principio porta a codice più manutenibile e, in linea di massima, più 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 (intuitivamente, una classe gestore per ogni tipologia distinta di risorsa) e focalizzarsi sull'uso appropriato delle risorse.

[Torna all'indice](#)

---

## OCP (Open-Closed Principle)

Il principio "aperto-chiuso" è forse il più conosciuto dei principi SOLID. Ne sono state proposte due varianti: la prima si fa risalire al lavoro di Bertrand Meyer (1988); la seconda (che è quella adottata nei principi SOLID) fu proposta da Robert C. Martin nel 1996, ma è di fatto una riformulazione di principi di progettazione proposti molti anni prima sotto altri nomi.

Usando le parole di Martin: "*SOFTWARE 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 fare ciò sia necessario modificare il codice esistente (chiusura alle modifiche).

Si noti che, nella sua enunciazione, il principio OCP non fornisce una metodologia esplicita per ottenere gli obiettivi che si propone, ragione per cui potrebbe sembrare inutile. In ogni caso, il principio può servire come linea guida per valutare quale, tra diverse alternative di progetto, soddisfa meglio i requisiti di apertura (alle estensioni) e chiusura (alle modifiche). In realtà, nei lavori che trattano del principio OCP, si dice esplicitamente che l'operazione chiave per ottenere un progetto aderente al principio suddetto è 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: usiamo il principio OCP per confrontare le due varianti di progetto (old-style vs oo-style) del nostro esercizio Fattoria.

Iniziamo con il valutare quali parti del codice saranno (probabilmente) oggetto di cambiamento: come si era detto informalmente quando si era introdotto il problema, si prevede l'introduzione di nuovi animali, con caratteristiche più o meno diverse da quelli esistenti, che dovranno comunque essere utilizzabili nel "codice utente" (il codice che produce le strofe della canzoncina). Dobbiamo quindi valutare se il codice è aperto alle estensioni (aggiunta di nuovi animali) pur restando chiuso alle modifiche (il codice che genera la strofa della canzone e il codice degli animali preesistenti non dovrebbero essere modificati in seguito all'introduzione di nuovi animali).

[Torna all'indice](#)

### Variante old-style

Nella variante old-style, esiste una unica classe Animale che implementa tutti gli animali concreti: l'aggiunta di un nuovo animale comporta la modifica di questa classe e, di conseguenza, la possibilità di modificare (magari inavvertitamente) il comportamento di uno degli animali preesistenti. In altre parole, il codice è aperto alle estensioni, ma è solo parzialmente chiuso alle modifiche: pur essendo vero che non dobbiamo modificare il codice che genera la strofa, ogni estensione che aggiunge un animale rischia di rompere il codice preesistente (inoltre, il codice che genera la strofa va comunque ricompilato, perché dipende direttamente dai dettagli implementativi della classe Animale).

[Torna all'indice](#)

## Variante oo-style

Nella variante oo-style, invece, abbiamo una classe astratta Animale; questa fornisce l'interfaccia, ma non mostra alcun dettaglio implementativo. Il codice è aperto alle estensioni: per aggiungere un animale è sufficiente creare una nuova classe che implementa (mediante derivazione pubblica "IS-A") l'interfaccia astratta. Il codice è anche chiuso alle modifiche, perché queste aggiunte non hanno nessun impatto sul codice che genera la strofa e nemmeno sulle classi che implementano tutti gli altri animali. Quando si aggiunge un animale, non c'è nemmeno bisogno di ricompilare il codice che genera la strofa: deve essere solo ricollegato. In linea di principio, team di sviluppatori diversi potrebbero generare varianti diverse degli animali, senza dovere condividere il codice sorgente (solo l'interfaccia astratta).

Quindi, fissato il tipo di modifica "aggiunta di nuovi animali concreti", possiamo affermare che il progetto oo-style soddisfa il principio OCP in misura maggiore rispetto al progetto old-style.

In entrambi i casi, l'aggiunta di nuovi animali concreti prevede piccole modifiche al modulo Maker.cc, per consentire ai nuovi animali di essere utilizzati dal programma. In altre parole, la "chiusura alle modifiche" non può mai essere totale: se vogliamo l'estendibilità devono sempre esistere dei punti in cui questa viene resa possibile; chiaramente, un buon progetto dovrebbe "confinare" il codice soggetto a modifiche in una zona ben delimitata e, per quanto possibile, piccola.

[Torna all'indice](#)

---

## DIP (Dependency 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."*

Il principio opera una classificazione sulle dipendenze tra moduli software (classi, funzioni, ecc.), stabilendo che alcune di queste dipendenze sono ammesse (in quanto inevitabili e tutto sommato innocue), mentre altre dipendenze sono da evitare (in quanto dannose).

Intuitivamente, le dipendenze "buone" sono quelle verso i concetti astratti; le dipendenze "cattive" sono quelle verso i dettagli implementativi.

Rimane da capire come mai il nome del principio parla di "inversione" delle dipendenze: è un punto importante, in quanto mette in evidenza l'aspetto metodologico nello sviluppo del software.

L'osservazione chiave è che, molto spesso, il software viene progettato e sviluppato seguendo un approccio top-down: partendo dal problema generale da risolvere, lo si suddivide in sottoproblemi più piccoli; la soluzione del problema generale si ottiene effettuando una opportuna composizione delle soluzioni dei sottoproblemi. Il processo viene ripetuto sui sottoproblemi, arrivando ad una stratificazione del codice, nella quale i moduli a livello più alto usano (e quindi *dipendono da*) i moduli a livello più basso. Si creano quindi naturalmente delle dipendenze (dall'astratto verso il concreto) che il principio DIP classifica come "cattive". Il principio DIP suggerisce quindi di "invertire" queste dipendenze, sostituendole con altre che invece non creano problemi (perché vanno dal concreto verso l'astratto).

A tale scopo, si individano alcune interfacce astratte, che non dipendono dai dettagli implementativi: **i moduli di alto livello vengono modificati per usare** (dipendere da) **le interfacce astratte**; analogamente, **i moduli a basso livello vengono modificati per implementare** (dipendere da) **le interfacce astratte** (realizzando quindi l'inversione). Complessivamente, si è eliminata la dipendenza dei moduli a alto livello dai moduli a basso livello. In particolare, si è migliorata anche l'aderenza del progetto al principio OCP, in quanto è ora possibile estendere il software, per esempio, consentendo la scelta di implementazioni alternative dell'interfaccia astratta senza influenzare i moduli ad alto livello.

Siccome il DIP si concentra sulle dipendenze tra moduli e queste dipendenze sono spesso rispecchiate dalla suddivisione in file del software (in particolare, dalle inclusioni di header file), a volte si dice che il principio DIP può essere visto come una reinterpretazione del principio OCP che si concentra sugli aspetti "sintattici".

[Torna all'indice](#)

# LSP (Liskov Substitution Principle)

Questo principio prende il nome da Barbara Liskov, che nel 1987 aveva enunciato la nozione di "sostituibilità" per tipi di dato (nozione poi formalizzata in un articolo scritto insieme a Jeannette Wing).

Intuitivamente, si dice che S è un sottotipo di T se ad ogni modulo che usa un oggetto t di T è 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 modo abbastanza grossolano) 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."*

Più propriamente, quello che si sta definendo è il cosiddetto "behavioral subtyping": le classi derivate (il sottotipo S) devono soddisfare le legittime aspettative degli utenti che accedono ad esse usando puntatori o riferimenti alle classi astratte (il tipo T). In altre parole, siccome S dichiara di essere in relazione "IS-A" rispetto a T, gli oggetti di tipo S non solo devono fornire (sintatticamente) i metodi forniti dalla classe base T, ma si devono anche comportare (behavior, aspetto semantico) come se fossero degli oggetti di tipo T.

Quindi il principio LSP può essere visto come una reinterpretazione del principio OCP che si focalizza sugli aspetti semantici (in precedenza avevamo notato che il DIP si focalizza solo sugli aspetti sintattici).

La corrispondenza del behavior non deve però essere intesa in senso assoluto (cioè, S non deve necessariamente essere identico a T): essa è limitata a quelle che sono le "aspettative legittime" che può avere un utente della classe T. Quali sarebbero queste legittime aspettative? Sono quelle stabilite dal *contratto* (precondizioni, invarianti e postcondizioni) che la classe T ha sottoscritto con i suoi utenti. Il principio LSP (e il behavioral subtyping) sono quindi in connessione stretta con la programmazione per contratto. Quando la classe derivata S dichiara di essere in relazione "IS-A" con la classe base T, di fatto si impegna a rispettare il contratto che T ha stabilito con i suoi utenti.

Esempio: consideriamo l'esempio della Fattoria. Un animale concreto soddisfa il principio LSP se, quando implementa i tre metodi virtuali dell'interfaccia astratta, rispetta il contratto stabilito da questa con i suoi utenti. Il contratto, purtroppo, non è stato stabilito esplicitamente, ma esiste e va rispettato, altrimenti l'utente si troverebbe di fronte a comportamenti erronei. Per esempio, è stato detto che il nome dell'animale deve essere il suo "nome comune" e che il genere (che serve a stabilire l'articolo indeterminativo da usare nella strofa) deve corrispondere al genere del nome comune (es., maschile per il cane, femminile per la volpe). Una classe concreta che, invece, fornisce il genere dell'animale (es., femminile per un cane femmina e maschile per una volpe maschio) violerebbe il principio LSP (e il contratto della classe base Animale).

Pur non essendo frequenti, le violazioni del principio LSP sono pericolose, perché non esistono modi semplici per rilevarle.

Un esempio classico di violazione del principio LSP è il seguente. Supponiamo che esista una classe Rettangolo, con la seguente interfaccia:

```
class Rettangolo {
    long lung;
    long largh;

public:
    bool check_inv() const {
        return lung > 0 && larg > 0;
    }

    Rettangolo(long lunghezza, long larghezza) :
        lung(lunghezza), larg(larghezza) {
            if (!check_inv())
                throw std::invalid_argument("Dimensioni invalide");
    }

    long get_lunghezza() const { return lung; }
    long get_larghezza() const { return larg; }

    void set_lunghezza(long value) {
        if (value <= 0)
            throw std::invalid_argument("Lunghezza minima");
    }
}
```

```

        throw std::invalid_argument("Dimensione invalida");
        lung = value;
    }
void set_larghezza(long value) {
    if (value <= 0)
        throw std::invalid_argument("Dimensione invalida");
    larg = value;
}

long get_area() const { return lungh * larg; }
};

```

Ad un certo punto viene richiesto di creare la classe Quadrato, dotata di una interfaccia simile. Siccome un quadrato è un tipo particolare di rettangolo, un programmatore (pigro) potrebbe pensare di implementare la classe quadrato usando l'ereditarietà pubblica e il polimorfismo dinamico. Le uniche accortezze tecniche sono quelle di dichiarare i metodi di Rettangolo come virtual (non puri, in quanto è una classe base concreta), di aggiungere il distruttore virtual e, infine, di fare l'override dei metodi `set_lunghezza` e `set_larghezza`, per assicurarsi che quando si modifica una dimensione sia modificata anche l'altra, così da mantenere l'invariante della classe Quadrato.

```

class Quadrato : public Rettangolo {
public:
    bool check_inv() const {
        return lung > 0 && lung == larg;
    }

    Quadrato(long lato) : Rettangolo(lato, lato) { }

    void set_lunghezza(long value) override {
        Rettangolo::set_lunghezza(value);
        Rettangolo::set_larghezza(value);
    }
    void set_larghezza(long value) override {
        set_lunghezza(value);
    }
};

```

Questo progetto viola il principio LSP. In particolare, NON è vero che a qualunque funzione che usa puntatori/riferimenti alla classe base (Rettangolo) noi possiamo passare invece puntatori/riferimenti alla classe derivata (Quadrato) e soddisfare le legittime aspettative dell'utente. In altre parole, NON è vero che un Quadrato "IS-A" Rettangolo, in quanto pur avendo la stessa interfaccia, non è equivalente dal punto di vista semantico.

Esempio: l'utente che usa la classe Rettangolo si aspetta che questo codice (in assenza di overflow) sia corretto, ovvero che l'asserzione sia sempre soddisfatta:

```

void raddoppia_area(Rettangolo& r) {
    long a_prima = r.get_area();
    long i = r.get_lunghezza();
    r.set_lunghezza(2 * i); // raddoppia la lunghezza
    long a_dopo = r.get_area();
    assert(a_dopo == 2 * a_prima);
    // .. altro codice
}

```

Se però alla funzione viene passato un riferimento a un Quadrato, il metodo `set_lunghezza()` raddoppierebbe sia la lunghezza che la larghezza e, di conseguenza, avremo una violazione dell'asserzione (perché `a_dopo == 4 * a_prima`).

[Torna all'indice](#)

## Cosa è successo?

Semplicemente, la classe Quadrato ha violato il contratto (stabilito dalla classe Rettangolo) del metodo `set_lunghezza`. Il contratto stabilisce (nelle sue post-condizioni) che il metodo modifica *solo* la lunghezza del rettangolo, mentre l'overriding definito nella classe Quadrato modifica sia la lunghezza che la larghezza.

Si potrebbe obiettare: ma un quadrato deve avere i lati uguali. L'obiezione è sensata, ma sta ad indicare che la classe Quadrato NON può essere in relazione IS-A con la classe Rettangolo, ovvero che un Quadrato NON è un Rettangolo. Quando facciamo questa affermazione, chiaramente, non stiamo ragionando in puri termini geometrici, ma stiamo piuttosto considerando l'aspetto "behavioral" dei corrispondenti tipi di dato: un Quadrato non è un Rettangolo perché

esistono dei contesti (vedi la funzione di sopra) in cui un Quadrato NON si comporta come si comporterebbe un Rettangolo.

La classe base stabilisce un contratto per ognuno dei suoi metodi, in termini di pre-condizioni e post-condizioni. Cosa deve fare la classe derivata per soddisfare tale contratto? Non è necessario che il contratto stabilito dalla classe derivata sia identico, ma la classe derivata deve fornire come minimo tutte le garanzie fornite dalla classe base. Quindi, la classe derivata può *indebolire* le precondizioni (cioè fornire una implementazione per più casi rispetto a quelli previsti dalla classe base) e può *rafforzare* le postcondizioni (cioè fornire all'utente garanzie ulteriori oltre a quelle garantite dalla classe base). Nel caso analizzato nell'esempio, le post-condizioni dei metodi `set_lunghezza` e `set_larghezza` sono state modificate (rendendole incompatibili, non rafforzarle), da cui la violazione del contratto e, di conseguenza, del principio di sostituibilità di Liskov.

[Torna all'indice](#)

---

## 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. Di conseguenza, il progettista di una interfaccia dovrebbe fare il possibile per *separare* quelle porzioni che potrebbero essere usate separatamente le une dalle altre, ovvero a preferire tante interfacce "piccole" (thin interfaces) rispetto a poche interfacce "grandi" (fat interfaces).

Aderendo a questo principio, si ottengono i seguenti benefici:

1. l'implementatore 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 è 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.

Dal punto di vista tecnico, l'applicazione del principio ISP presuppone la possibilità di utilizzare l'ereditarietà multipla (di interfaccia) e, in effetti, può essere considerato l'esempio più frequente di uso appropriato dell'ereditarietà multipla. E' quindi importante studiare i corrispondenti aspetti tecnici dal punto di vista del linguaggio.

Si noti infine che il principio ISP può essere interpretato come una forma particolare del principio SRP, che si concentra al caso specifico della progettazione delle interfacce.

[Torna all'indice](#)

# Alcune questioni tecniche sul polimorfismo dinamico

Abbiamo visto che, nel caso di polimorfismo dinamico, le classi astratte sono tipicamente formate da metodi virtuali puri, più il distruttore della classe che è dichiarato virtuale, ma non puro. In alcuni casi è però necessario complicare il progetto (ad esempio, usando l'ereditarietà multipla): quando lo si fa, si corre il rischio di incorrere in errori ed è quindi 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 metodi 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
    - distruttori
    - funzioni membro (di istanza, cioè non statiche)
    - funzioni membro statiche
    - template di funzioni membro (non statiche)
    - funzioni membro (non statiche) di classi templatiches
  2. Come faccio a costruire una copia di un oggetto concreto quando questo mi viene fornito come puntatore/riferimento alla classe base?
  3. Cosa succede se si invoca un metodo virtuale durante la fase di costruzione o di distruzione di un oggetto?
  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
5. Quali sono gli usi del polimorfismo dinamico nella libreria standard?
    - classi eccezione standard
- classi iostream

## Metodi che non possono essere `virtual`

### Costruttori (NO)

Al momento della creazione, **non esiste ancora** un oggetto `this` **interrogabile** (esiste ma lo si sta definendo in quel momento).

### Distruttori (SI, tipicamente obbligatorio)

Vogliamo eliminare correttamente l'oggetto nella sua interezza; inoltre deve essere implementato in quanto verrà sicuramente invocato nella catena di chiamate ai distruttori (dall'oggetto di tipo `Derived` a quello `Base`).

```
struct A {  
    ~A() {}  
};  
  
struct B : public A {  
    ~B() {}  
};  
  
{  
    A* pa_b = new B;  
    delete pa_b; // errore: non chiama ~B()  
}
```

### Funzioni membro (SI)

Sono le classiche funzioni membro di una classe standard.

## Funzioni membro statiche (NO)

Sono funzioni che fanno riferimento alla classe, non hanno il puntatore `this`, non c'è nessun modo che permetta al RTTI (*Run Time Type Identification*) di effettuare una qualche forma di dispatching della funzione in questione.

## Template di funzioni membro, non statiche (NO)

Il `this` esiste, quindi in linea teorica sarebbe legittimo dichiararle virtuali; si è deciso però di non implementare questa possibilità, per efficienza. Facciamo un esempio:

```
struct F {  
    virtual void do() = 0; // OK  
  
    template  
    virtual void foo() = 0; // Errore  
};
```

Così facendo, l'utente potrebbe istanziare un numero grande a piacere di funzioni virtuali `foo()`, il che significherebbe una grandezza arbitraria della **V-Table** (tabella dei metodi virtuali).

## Funzioni membro (non statiche) di classi templatiche (SI)

Una volta scelto il tipo templatico della classe in questione, l'insieme delle funzioni virtuali è finito e numerabile, quindi è ammissibile:

```
template  
class Animale {  
public:  
    virtual void verso() = 0;  
    virtual void foo(T& t) = 0;  
};  
  
// Animale a; --> ci sono 2 funzioni virtuali
```

---

## Copia di un oggetto concreto

Intuitivamente serve un metodo virtuale che costruisca correttamente un nuovo oggetto a partire dal riferimento/puntatore passato: il metodo `clone()`.

```
class Animale {  
    virtual void verso() const = 0;  
    virtual Animale* clone() const = 0;  
};  
  
class Cane : public Animale {  
    void verso override() const { /* */ }  
    Cane* clone() const override { return new Cane(*this); }  
};  
  
void foo(const Animale* pa) {  
    Animale* pa_2 = pa->clone();  
    // ...  
}
```

Osserviamo che nell'implementazione del metodo di clonazione, il tipo restituito può essere cambiato con quello della classe derivata (`Cane*` invece di `Animale*`, eccezione del linguaggio), evitando *down-casting* esplicito. Questi metodi vengono detti **costruttori virtuali**.

---

## Invocazione di funzione virtuale in un costruttore/distruuttore

La risoluzione dell'overriding in questo caso specifico funziona in maniera differente. Fino al C++98 si adottava l'approccio classico, ed era possibile incorrere in evidenti problematiche: al momento della costruzione di un oggetto Derived, se il costruttore di Base facesse una chiamata ad un metodo virtuale, ci sarebbe accesso a memoria non ancora

inizializzata e possibile *undefined behaviour*:

```
struct Base {
    Base() {
        foo();
    }
    virtual void foo () { std::cout << "Base::foo()" << std::endl; }
};

struct Derived : public Base {
    int* p_i;

    Derived() : p_i(new int) {
        *p_i = 7;
        foo();
    }
    virtual ~Derived() { delete p_i; }
    virtual void foo() { std::cout << *p_i << std::endl; }
};

{
    Derived d;
}
```

Il costruttore di `Derived` chiama implicitamente quello di `Base`, la chiamata a `foo()` viene risolta con `Derived::foo()` andando quindi ad effettuare una dereferenziazione di un puntatore non ancora inizializzato.

Dallo standard C++11 il supporto a tempo di esecuzione cambia **incrementalmente** il tipo dinamico dell'oggetto in questione: al momento della costruzione della parte `Base` di un oggetto di tipo `Derived`, quello stesso oggetto viene considerato di tipo `Base` e viene invocata correttamente `Base::foo()`; a questo punto, il tipo dinamico dell'oggetto cambia in `Derived` e viene invocata `Derived::foo()`.

Stampa:

```
Base::foo();
7
```

Nei distruttori si ha lo stesso comportamento.

---

## Ereditarietà multipla con classi non interfaccia

Utilizzando più classi base con lo stesso campo, potrebbero esserci oggetti ripetuti:

```
struct B1 {
    int a;
};

struct B2 {
    int a;
};

struct D : public B1, public B2 {
    void foo() {
        std::cout << a << std::endl; // ambiguo, dovrei specificare B1::a o B2::a
    }
};
```

Supponiamo ora un altro caso:

```
struct B0 {
    int a;
};

struct B1 : public B0 {
};

struct B2 : public B0 {
};

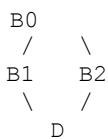
struct D : public B1, public B2 {
};
```

È possibile avere una struct `B0` condivisa tra `B1` e `B2` nella struct `D`? Si, utilizzando l'ereditarietà di tipo `virtual`.

```
struct B1 : virtual public B0 {  
};  
struct B2 : virtual public B0 {  
};
```

Si genera uno schema *a diamante* in cui la classe `D` deriva da altre due classi che a loro volta derivano comunemente da `B0`.

Conosciuto come *DDD* problem (Dreadful Diamond on Derivation)



Nella **costruzione** di un oggetto di tipo `D`, la **semantica di costruzione è diversa**: prima le classi base virtuali (in questo caso `B0`) nell'ordine di visita del grafo di ereditarietà, successivamente tutte le classi base che non sono virtuali (`B1` e `B2`) evitando di re-inizializzare quelle virtuali; infine si procede con la costruzione dell'oggetto derivato.

La stessa semantica è applicata nella **distruzione**, in ordine inverso: distruggo l'oggetto derivato, successivamente `B1` e `B2` senza toccare `B0` e infine il distruttore di `D` si occupa di chiamare quello di `B0`.

# Programmazione per contratto

---

## pre-condizioni e post-condizioni

Una corretta definizione dell'interfaccia di una classe prevede la stesura di una sorta di contratto tra lo sviluppatore della classe e l'utilizzatore della classe. Per ogni funzionalità fornita, il contratto stabilisce quali sono le *pre-condizioni* che l'utilizzatore deve soddisfare per potere invocare la funzionalità e quali sono le *post-condizioni* che l'implementatore deve garantire in seguito all'esecuzione della funzionalità.

Tra le pre-condizioni e le post-condizioni sono sempre incluse le invarianti di classe sugli oggetti che sono acceduti (in lettura e/o scrittura) durante l'implementazione della funzionalità.

Il contratto si specifica in questo modo:

```
$$ \text{pre-condizioni} \rightarrow \text{post-condizioni} $$
```

Si noti che se le pre-condizioni NON sono valide (cioè sono false), allora l'implicazione del contratto è vera a prescindere dalla validità o meno delle post-condizioni. Ovvero, se l'utente non soddisfa una pre-condizione l'implementatore non ha alcun obbligo. Spesso si preferisce rendere esplicite le condizioni sulle invarianti di classe, scrivendo il contratto nella forma:  $\text{pre-condizioni AND invarianti} \rightarrow \text{post-condizioni AND invarianti}$ . In questo caso, pre-condizioni e post-condizioni sono intese "al netto" delle invarianti di classe. Per esempio, nel caso dell'operatore di divisione tra oggetti Razionale:

```
Razionale operator/(const Razionale& x, const Razionale& y) {
    assert(x.check_inv() && y.check_inv()); // invarianti in ingresso
    assert(y != Razionale(0)); // pre-condizione "al netto" delle invarianti

    Razionale res = x;
    res /= y;

    // invarianti in uscita (omesso per x e y, perché costanti)
    assert(res.check_inv());
    return res;
}
```

La pre-condizione "al netto delle invarianti" dice che l'oggetto y deve essere diverso dal Razionale zero.

Analogamente, la post-condizione "al netto delle invarianti" richiede che il valore restituito sia effettivamente il risultato della divisione di x per y: i controlli sulle post-condizioni sono spesso difficili se non impossibili da automatizzare all'interno della classe e quindi si codificano direttamente nei test, indicando i risultati attesi.

[Torna all'indice](#)

---

## Contratti narrow

Un operatore di divisione come specificato sopra è un esempio di "contratto narrow" (stretto): nei contratti narrow, l'implementatore si impegna a fornire la funzionalità solo quando ha senso farlo, cioè quando i valori forniti in input sono legittimi; l'onere di verificare tale legittimità è lasciato all'utilizzatore.

I contratti di tipo narrow sono molto comuni in C++, sia nel linguaggio in senso stretto (per esempio, quando si accede ad un elemento di un array, l'onere di controllare la validità dell'indice è a carico del programmatore), sia a livello di libreria standard (per esempio, spetta all'utente controllare che un `std::vector` non sia vuoto prima di eliminare l'ultimo elemento usando il metodo `pop_back`).

[Torna all'indice](#)

---

## Contratti wide

Un caso ben diverso si verifica nel caso dei "contratti wide" (ampi), nei quali l'onere di verificare la legittimità delle invocazioni ricade sull'implementatore. Scegliere un contratto wide equivale quindi a spostare alcuni elementi del contratto dal lato della pre-condizione al lato della post-condizione.

Esempio di operatore di divisione con contratto wide:

```
Razionale operator/(const Razionale& x, const Razionale& y) {
    assert(x.check_inv() && y.check_inv()); // invarianti in ingresso

    // il contratto wide prevede, come post-condizione, che nel caso
    // y sia zero venga lanciata una opportuna eccezione
    if (y == Razionale(0))
        throw DivByZero();

    Razionale res = x;
    res /= y;

    // invarianti in uscita (omesso per x e y, perché costanti)
    assert(res.check_inv());
    return res;
}
```

Notare come, in caso di contratto wide, il controllo che y sia diverso da zero deve essere esplicito (NON può essere implementato come asserzione, perché deve essere presente anche nel codice compilato in modalità di produzione); inoltre, anche se la condizione viene controllata all'inizio dell'implementazione dell'operatore, si tratta di una *post* condizione, perché in questo caso l'utente può legittimamente pretendere di ottenere l'eccezione `DivByZero` quando y è uguale a zero.

I contratti wide sono quindi più onerosi (sia in termini di efficienza che in termini di codice da scrivere) per l'implementatore della classe.

[Torna all'indice](#)

---

## I contratti per il linguaggio C++ e la libreria standard

Ogni volta che lo standard descrive una funzionalità del linguaggio o della libreria standard, ne viene descritto (a volte implicitamente) il contratto, in termini di pre-condizioni e post-condizioni. In particolare, sono classificati i comportamenti (behaviors) che una specifica implementazione del linguaggio è tenuta a rispettare, distinguendo tra le seguenti categorie:

### 1. Comportamento specificato (specified behavior)

Il comportamento è descritto dallo standard; ogni implementazione è tenuta a conformarsi alla prescrizione, come specificata.

### 2. Comportamento definito dall'implementazione (implementation-defined)

Ogni implementazione può scegliere come realizzare una determinata funzionalità, con l'obbligo di documentare la scelta fatta; per esempio, la scelta della dimensione di ognuno dei tipi interi standard, oppure la scelta se il tipo "char" (plain, cioè senza l'uso di signed o unsigned) abbia o meno il segno.

### 3. Comportamento non specificato (unspecified behavior)

Comportamento che dipende dall'implementazione, che però non è tenuta a documentare la scelta fatta (e nemmeno a rifare la stessa scelta in contesti diversi); ad esempio, l'ordine di valutazione delle sottoespressioni è non specificato (caso particolare, l'ordine di valutazione degli argomenti in una chiamata di funzione).

### 4. Comportamento non definito (undefined behavior)

Comportamento ottenuto a causa della violazione di una pre-condizione, in seguito alla quale l'implementazione non ha più nessuna prescrizione da seguire e quindi potrebbe comportarsi in modo totalmente arbitrario; esempi: indicizzazione di un array al di fuori dei limiti; tentativo di scrivere su di un oggetto definito `const`; overflow sui tipi interi con segno, ecc. Viene spesso indicato con la sigla UB e, per sottolinearne la totale arbitrarietà descritto,

colloquialmente, con la possibilità "to make demons fly out of your nose".

Esiste anche il locale-specific behavior, che dipende da convenzioni, cultura o linguaggio; per esempio, il valore dei caratteri del cosiddetto "execution character set".

[Torna all'indice](#)

# Definizioni

---

## Namespaces

In C++, i namespaces sono **utilizzati per organizzare il codice in gruppi logici**, allo scopo di evitare conflitti di nomi e migliorare la leggibilità del codice.

Un namespace è un'area di memoria in cui vengono memorizzati nomi di variabili, funzioni, classi e altre entità di programmazione. Un namespace può contenere altre entità di namespace, creando così una gerarchia di namespace.

Ad esempio, si può creare un namespace per le funzioni matematiche, come segue:

```
namespace math {  
    double pi = 3.14159265358979323846;  
  
    double radice_quadrata(double x) {  
        return sqrt(x);  
    }  
}
```

In questo esempio, abbiamo creato un namespace chiamato `math`. All'interno di questo namespace, abbiamo definito una costante chiamata `pi` e una funzione chiamata `radice_quadrata`.

Per utilizzare le entità definite in un namespace, è necessario specificare il nome del namespace prima del nome dell'entità. Ad esempio, per utilizzare la costante `pi` definita nel namespace `math`, si può scrivere:

```
double raggio = 5.0;  
double circonferenza = 2 * math::pi * raggio;
```

In questo esempio, stiamo utilizzando il nome del namespace `math` per accedere alla costante `pi` definita all'interno del namespace.

L'utilizzo dei namespaces può **rendere il codice più leggibile**, in quanto evita conflitti di nomi tra le entità di programmazione. Ad esempio, si può utilizzare lo stesso nome di funzione in due namespace diversi, senza generare conflitti tra i nomi.

Inoltre, l'utilizzo dei namespaces consente di utilizzare librerie esterne e di integrare codice da diverse fonti, senza generare conflitti tra i nomi delle entità di programmazione.

In generale, l'utilizzo dei namespaces è una pratica utile per organizzare il codice in modo coerente e migliorare la leggibilità del codice.

[Torna all'indice](#)

---

## Templates

In C++, i templates sono una **funzionalità che consente di creare codice generico e riusabile**. In pratica, un template definisce un modello o un prototipo per una funzione o una classe, consentendo di creare versioni specializzate di tali funzioni o classi per diversi tipi di dati.

Ad esempio, si può creare un template per una funzione che restituisce il valore massimo tra due valori di un determinato tipo di dati:

```
template  
T max(T a, T b) {  
    return a > b ? a : b;  
}
```

In questo esempio, abbiamo creato una funzione chiamata `max` utilizzando la sintassi del template. La parola chiave `template` viene utilizzata per definire il template, seguita dal nome del tipo di dati generico `T`. La funzione `max` prende

due argomenti di tipo `T` e restituisce il valore massimo tra di essi.

Si può utilizzare questa funzione per trovare il massimo tra due numeri interi o tra due numeri in virgola mobile, senza dover scrivere due funzioni diverse:

```
int a = 10, b = 20;
cout << max(a, b) << endl;

double x = 3.14, y = 2.71;
cout << max(x, y) << endl;
```

In questo esempio, stiamo utilizzando la funzione `max` per trovare il massimo tra due numeri interi e tra due numeri in virgola mobile.

I templates sono utili perché consentono di scrivere codice generico e riusabile, che può essere utilizzato con diversi tipi di dati. Inoltre, i templates consentono di scrivere codice più leggibile e mantenibile, poiché il codice ripetitivo può essere eliminato utilizzando il template.

In generale, l'utilizzo dei templates è una pratica utile per creare codice generico e riusabile in C++.

[Torna all'indice](#)

---

## Funzioni inline

Le funzioni `inline` in C++ sono utilizzate per richiedere al compilatore di sostituire il codice della funzione al punto in cui viene chiamata. Ciò può migliorare le prestazioni del programma, in quanto evita il tempo di esecuzione richiesto per la chiamata della funzione.

In C++, le *funzioni inline* possono essere definite in due modi:

1. **Dichiarazione inline:** La parola chiave `inline` viene utilizzata davanti alla dichiarazione della funzione, come nel seguente esempio:

```
inline int somma(int a, int b) {
    return a + b;
}
```

2. **Definizione inline:** La definizione della funzione viene inserita direttamente nel corpo della classe, dichiarando la funzione come `inline` implicitamente. Questo metodo viene spesso utilizzato per le funzioni membro di classe, ma può essere utilizzato anche per le funzioni globali, come nel seguente esempio:

```
class Calcolatrice {
public:
    inline int somma(int a, int b) {
        return a + b;
}
};
```

In entrambi i casi, l'utilizzo di una funzione `inline` è un suggerimento al compilatore per sostituire il codice della funzione al punto in cui viene chiamata, ma il compilatore non è obbligato a farlo. In altre parole, il compilatore può ignorare la richiesta di `inline` se lo ritiene necessario.

## Vantaggi

Ci sono alcuni vantaggi nell'utilizzo delle funzioni inline in C++. In primo luogo, il tempo di esecuzione richiesto per la chiamata di una funzione viene eliminato, il che può migliorare le prestazioni del programma. In secondo luogo, l'utilizzo di funzioni inline può ridurre la dimensione del codice generato dal compilatore, in quanto il codice della funzione viene inserito direttamente nel punto in cui viene chiamata, anziché generare codice per la chiamata della funzione.

## Svantaggi

Tuttavia, ci sono anche alcuni svantaggi nell'utilizzo delle funzioni `inline`. Ad esempio, l'utilizzo di funzioni `inline` può aumentare la dimensione del codice generato dal compilatore, in quanto il codice della funzione viene ripetuto in ogni punto in cui viene chiamata. Inoltre, l'utilizzo di funzioni `inline` può aumentare la complessità del codice, in quanto il compilatore deve gestire la sostituzione del codice della funzione al punto in cui viene chiamata.

In generale, le funzioni `inline` dovrebbero essere utilizzate con cautela, e dovrebbero essere applicate solo alle funzioni che vengono chiamate frequentemente e che sono relativamente brevi. Ciò può aiutare a migliorare le prestazioni del programma, senza aumentare la dimensione del codice o la complessità.

==TODO==

- `constexpr`
- `#define PROD(a,b) (a) * (b)`

[Torna all'indice](#)

---