MdP completo

1 - Processo di compilazione

Il processo di compilazione prende in input file sorgente e/o librerie e produce in output file eseguibili o librerie, i passaggi sono:

1. Il processore elabora il codice del file sorgente per produrre una **unità di traduzione**:
   `g++ -E hello.cpp -o hello.preproc.cpp`
   la dimensione dell'unità di traduzione è particolarmente più estesa rispetto a quella degli altri file in quanto, il prepocessore include il codice di molti altri file sorgente (es. `<iostream>`).
2. il compilatore elabora l'unità e produce codice assembler:
   `g++ -Wall -Wextra -S hello.cc -o hello.s : -Wall -Wextra`
3. l'assemblatore produce da questo il file oggetto:
   `g++ .c hello.s -o hello.o : -c`
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):
   `g++ hello.o -o hello`

2 - Dichiarazioni e definizioni

Dichiarazione: costrutto che introduce (dichiara) un nome per una entità. Informa il compilatore del tipo e delle proprietà di una variabile o di una funzione, ma senza riservare spazio in memoria o fornire dettagli sull'implementazione.
Definizioni: dichiarazione che, oltre al nome, fornisce ulteriori elementi per caratterizzare l'entità. Oltre a dichiarare, assegna un valore nel caso di una variabile, o un'implementazione nel caso di una funzione e alloca memoria.

Tutte le definizioni sono anche dichiarazioni, ma non tutte le dichiarazioni sono definizioni, in questo caso vengono chiamate **dichiarazioni pure**.

Esempi:

1. *Tipi di dato*
   `Struct S;`
   dichiarazione pura del tipo S, non possiamo creare oggetti di questo tipo ma può tornare utile per la definizione di puntatori o riferimenti del tipo S, senza dover conoscere il tipo ⇒ **puntatori opachi**.
   `Struct T { int a; };`
   definizione del tipo T, possiamo creare oggetti di questo tipo.
2. *Variabili*:
   `extern int a`
   dichiarazione pura di variabile globale, il compilatore sa della sua esistenza ma la creazione verrà fatta successivamente
   `extern int d = 2`
   definizione, perché è inizializzata

3. *Funzioni*:
   `void foo (int a);`
   dichiarazione pura di funzione, è presente solo la signature della funzione
   `void foo(int a) { std::cout << a; }`
   definizione di funzione, viene fornita l'implementazione della funzione
4. *Template*:

```
template <typename T> struct S; // dichiarazione pura di template di classe
template <typename T>
struct S {
        T t;
}; // definizione di template di classe

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

3 - Scope

Le dichiarazioni introducono un nome per un entità, questo è **visibile** però solo in alcuni punti dell'unità di traduzione, la parte in cui è visibile viene chiamato **campo di azione (scope)**.
Tipologie:

- scope di namespace (incluso lo scope globale):
  - dichiarazione non racchiusa all'interno di una struct/class/enum o all'interno di una funzione
  - visibile dal punto di dichiarazione fino al termine dell'unità di traduzione, non visibile prima del punto di dichiarazione
- scope di blocco (codice racchiuso fra parentesi graffe):
  - un nome dichiarato in un blocco è locale a quel blocco
  - visibilità dal punto di dichiarazione fino alla fine del blocco
  - regole speciali per `for`, `while`, `if`, `switch`, `catch`; variabili dichiarate all'interno di questi costrutti hanno visibilità solo all'interno di essi.
- scope di classe:
  - i dati membro e i metodi di una classe/struct sono visibili all'interno della classe indipendentemente dal punto di dichiarazione, in modo tale da consentire la definizione di metodi inline.
  - per i tipi membro, valgono le regole dello scope di blocco
  - i membri di una classe possono essere acceduti da classi che sono derivate (perche ereditati) e possono essere acceduti dall'esterno:
    `s.foo(); //se s ha tipo S`
    `ps->foo(); //se ps ha tipo puntatore a S`
    `S::foo(); //operatore di scope`
    differenza tra class e struct:
    quasi nessuna differenza, soltanto l'accesso ai metodi e ai dati membro, in particolare quando

non vengono specificati gli access specified (public, private) nelle classi di default è tutto privato, mentre nelle struct di default è tutto pubblico.

- scope di funzione: riguarda soltanto le etichette (label) del `goto`

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

- scope delle costanti di enumerazione: con c++11 le `enum class` adottano le regole di scope delle classi, necessitando della qualificazione del nome, per non creare ambiguità, e del cast (necessario perche le enum class impediscono anche le conversioni implicite di tipo verso gli interi).

```
enum class Colori { rosso , blu , verde };
enum class Semaforo { verde , giallo , rosso };
void foo () {
std :: cout << static_cast<int>( Colori :: rosso );
}
```

Scope potenziale e Scope effettivo

Scope potenziale: come si comporterebbe se il programmatore non facesse nulla per andare a modificarlo (ridurlo o estenderlo).
Questo scope può essere modificato e diventa cosi **scope effettivo**.
Modifiche dello scope:

- **name hiding**: scope diversi vengono annidati e una dichiarazione nello scope interno può nascondere un'altra dichiarazione (con lo stesso nome) dello scope esterno. Si può avere hiding anche per i membri ereditati da una classe, perché lo scope della classe derivata è considerato incluso nello scope della classe base.
- **uso di nomi qualificati**: accesso ad alcune entità al di fuori del loro scope può essere ottenuto usando nomi qualificati (`std:endl`). La qualificazione può essere: parziale, punto di partenza lo scope corrente; totale, punto di partenza lo scope globale (FQN, Fully Qualified Name)
- **ADL: (Argument Dependent Lookup)**:
  in una chiamata di funzione `foo(.., arg, ..)` se il nome della funzione (`foo`) non è qualificato e se uno degli argomenti (`arg`) della chiamata è di un tipo dato `N::U` definito dall'utente all'interno

del namespace N, allora si considerano come candidate tutte le funzioni con lo stesso nome dichiarate all'interno dello stesso namespace (cioè `N:foo`)

- **dichiarazioni e direttive using**:
  - se un nome deve essere utilizzato spesso in una posizioni in cui non è visibile si utilizza la **using declaration** (`using std::cout`). Nel caso di una using declaration per un tipo o di una variabile, è necessario che nello stesso scopo non sia già presente un'altra entità con lo stesso nome. Questa cosa invece è legittima in caso di funzioni → **overloading**.
  - **using directive**: non introducce dichiarazioni nel punto in cui viene usata, ma aggiunge il namespace indicato tra gli scope nei quali è possibile cercare un nome (`using namespace std;`).

  N. B. :
- preferire le using declaration rispetto alle using directive, perche introducono meno nomi.
- limitare al massimo lo scope delle using declaration / directive
- non usarle a scope di namespace/globale, in particolare in un header file

4 - Lifetime

Mentre lo scope descrive la dimensione spaziale (dove) di un nome introdotto da una dichiarazione, il tempo di vita descrive la dimensione temporale, necessario per stabilire quando è legittimo interagire con una determinata entità. Alcune entità, come tipi di dato, funzioni, etichette, possono essere riferite in qualunque momento durante l'esecuzione del programma, mentre un oggetto memorizzato in memoria è utilizzabile solo dopo la sua creazione e soltanto fino a quando viene distrutto.

Creazione di oggetti in memoria

- definizione `int n = 42;` (non basta una dichiarazione pura;
- invocazione dell'espressione `new`, allocando oggetti nell'heap;
- valutazione di una espressione che crea implicitamente un nuovo oggetto (temporaneo e senza nome):

```
std::string str = "Hello ";
std::cout << ( str + "␣world!") << std::endl ;
```

Inizio e fine lifetime

- inizia quando un oggetto *termina* la sua **costruzione**, composta da due fasi:
  - allocazione della memoria "grezza";
  - inizializzazione della memoria quando prevista.
- termina quando un oggetto *inizia* la sua **distruzione**, composta da due fasi:
  - invocazione del distruttore (quando previsto);
  - deallocazione della memoria "grezza".

  Se la costruzione dell'oggetto non termina con successo, non ha iniziato il suo lifetime e non dovrà terminarlo.

  Durante le fasi di creazione e di distruzione dell'oggetto si è fuori dal suo ciclo di vita, perciò le operazioni consentite sono limitate

Tipologie di allocazione

Il lifetime di un oggetto dipende dal tipo di "allocazione" utilizzata per la creazione:

- allocazione statica
- allocazione thread local
- allocazione automatica
- allocazione automatica dei temporanei
- allocazione dinamica

Allocazione Statica

Il lifetime dell'oggetto dura "per tutta l'esecuzione del programma". Sono dotate di memoria ad allocazione statica:

- le variabili definite a namespace scope (dette globali):
  - sono create e inizializzate prima di iniziare l'esecuzione del main, nell'ordine in cui compaiono nell'unità di traduzione in cui sono definite;
  - **Static Inizialization Order Fiasco**: nel caso di variabili globali definite in diverse unità di traduzione, l'ordine di inizializzazione non è specificato causando errori;
  - in caso di terminazione normale del programa le globali sono distrutte dopo la terminazione della funzione main, in ordine inverso rispetto alla creazione.
- i dati membro di classi dichiarati usando `static`:
  stesse regole delle globali, tranne nel caso di template.
- le variabili locali (scope di blocco) dichiarate usando `static`:
  sono allocate prima di iniziare l'esecuzione del 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
  L'utilizzo dell'allocazione statica quando non strettamente necessario porta a codice poco leggibile e poco mantenibile. Se l'allocazione statica è necessaria, preferire i dati membro statici, se possibile dichiarate private, in modo da confinare il codice problematico.

Allocazione thread local

Un oggetto thread local è simile ad un oggetto globale, ma il suo ciclo di vita non è collegato al programma, bensi ad ogni singolo thread di esecuzione creato dal programma. Il supporto per il multithreading è stato introdotto con c++11.

Allocazione automatica

Una variabile locale ad un blocco di funzione (non dichiarata `static`) 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. Nel caso di funzioni ricorsive, sullo stack possono esistere contemporaneamente più istanze distinte della stessa variabile locale. L'oggetto viene poi automaticamente distrutto rimuovendolo dallo stack ogni volta che il controllo esce da quel blocco. L'inizio del lifetime è stabilito dal programmatore scrivendo la definizione della variabile locale, mentre la fine è automatica.

Allocazione automatica di temporanei

avviene quando un oggetto viene creato per memorizzare il valore calcolato da un sottoespressione che compare all'interno di una espressione, viene poi distrutto quando termina la valutazione

dell'espressione (1). Il lifetime del temporaneo può essere esteso se l'oggetto viene utilizzato per inizializzare un riferimento (2).

1.

```
struct S {
        S (int) { std::cout << "costruzione"; }
        ~S () { std::cout << "distruzione"; }
};
void foo ( S s );
void bar () {
        foo ( S (42)); // allocazione di un temporaneo per S (42)
        // temporaneo distrutto prima di stampare fine
        std::cout << " fine ";
}
```

2.

```
void bar2 () {
// il temporaneo `e usato per inizializzare il riferimento s
        const S & s = S (42);
        std::cout << "fine";
// il temporaneo `e distrutto quando si esce dal blocco ,
// dopo avere stampato " fine "
}
```

Allocazione dinamica

Un oggetto può essere allocato dinamicamente nella memoria heap con l'espressione `new`, che restituisce l'indirizzo dell'oggetto allocato, che va salvato in un opportuno puntatore. La distruzione non è automatica, ma è responsabilità del programmatore utilizzare l'espressione `delete` sul puntatore che contiene l'indirizzo dell'oggetto.

L'allocazione automatica e quella dei temporanei avviene dinamicamente, ovvero a tempo di esecuzione; la differenza sta nel fatto che la deallocazione è manuale.

Errori frequenti di programmazione:

- *memory leak*: accade quando il programmatore perde l'accesso al puntatore che riferisce l'oggetto allocato dinamicamente: la memoria dell'oggetto è ancora in uso, ma non può più essere letta, scritto o rilasciata
- *use after free*: l'utilizzo di un oggetto dopo che il suo lifetime è concluso, uso di un *puntatore dangling*
- *double free*: uso della delete più volte sullo stesso indirizzo, causando la distruzione di una porzione di memoria che non era più allocata e che potenzialmente è stata riutilizzata per altro.
- *accesso al null pointer*: si prova ad accedere ad un puntatore nulla (deferenziandolo)

5 - Tipi, qualificatori e costanti letterali

Tipi integrali

- tipi booleani `bool`
- tipi carattere:
  - narrow: `char`, `signed char`, `unsigned char`
  - wide: `wchar_t`, `char16_t`, `char32_t`
- tipi interi standard con segno: `signed char`, `short`, `int`, `long`, `long long`
- tipi interi senza segno: `unsigned char`, `unsigned short`, `unsigned int`...

Tipi integrali piccoli

Tipi boolenai, tipi caratteri narrow e short (con o senza segno) sono detti tipi integrali piccoli: potrebbero avere una dimensione (`sizeof`) inferiore al tipo `int` (non per forza).
I tipi integrali piccoli sono soggetti a una certa categoria di conversioni implicite: **promozioni**.

Tipi built-in non integrali

- Tipi floating point: `float`, `double`, `long double`
- Tipo void: insieme vuoto di valori
  - come tipo di ritorno di una funzione indica che non deve tornare alcun valore
  - usato nel cast esplicito, indica che il valore di una espressione deve essere scartato
    `(void) foo(3); //scarta il risultato di foo(3)`
- Tipo `std::nullptr_t` tipo puntatore convertibile implicitamente in qualunque altro tipo di puntatore; ha un solo valore possibile, la costante letterale `nullptr`, che indica il puntatore nullo (non deferenziabile).

Tipi composti

- `T&`: riferimento a lvalue T
- `T&&`: riferimento a rvalue T
- `T*`: puntatore a T
- `T[n]`: array
- `T(T1, T2, T3)`: tipo funzione
- enumerazioni, classi e struct

Tipi qualificati `const`

- **qualificatori di tipo**: `const` e `volatile`
- considerando solo il qualificatore `const`
  - essenziale per una corretta progettazione e uso delle interfacce software
  - utile come strumento di supporto
- dato un tipo T, è possibile fornire la versione qualificata `const T`
- l'accesso ad un oggetto attraverso una variabile dichiarata `const` è consentito in sola lettura
- nel caso di tipi composti è necessario distinguere tra le qualificazioni del tipo composto rispetto alla qualificazione delle sue componenti.

Costanti letterali

Vengono messe 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.

tipi costanti letterali

- *booleani*: `false`, `true`;
- *char*:
  - Si rappresentano racchiusi tra apici singoli: `'a'`, `'3'`.
    - Caratteri speciali come `'\n'` (nuova linea) possono essere inclusi.
    - Esistono diverse codifiche:
      - **UTF-8**: aggiungi `u8` davanti, esempio: `u8'a'`.
      - **UTF-16**: usa `u`, esempio: `u'a'`.
      - **UTF-32**: usa `U`, esempio: `U'a'`.
      - **Wide character** (più spazio per caratteri complessi): usa `L`, esempio: `L'a'`.
- *numeri interi*
  - Esempio base: `123`.
  - Puoi specificare varianti con **suffissi**:
    - `U`: per numeri senza segno (esempio: `123U`).
    - `L`: per numeri "grandi" (esempio: `123L`).
    - `LL`: per numeri ancora più grandi (esempio: `123LL`).
    - Puoi combinarli: `123UL` (unsigned long).
- *numeri a virgola mobile (floating point)*
  - Esempio base: `123.45`.
  - Puoi scriverli anche in **notazione scientifica**: `1.23e2` (che significa 1.23·1021.23 \cdot 10^21.23·102).
  - Usa i suffissi per specificare il tipo:
    - `F`: per `float` (esempio: `123.45F`).
    - `L`: per `long double` (esempio: `123.45L`).
- *stringhe*
  - Si scrivono tra virgolette doppie: `"Hello"`.
  - Il tipo associato è `const char[]` (un array di caratteri costanti).
  - Puoi aggiungere prefissi per cambiare la codifica:
    - `u8"Hello"`: UTF-8.
    - `u"Hello"`: UTF-16.
    - `U"Hello"`: UTF-32.
    - `L"Hello"`: Wide.
- *stringhe grezze (raw string literals)*
  - Sono stringhe che possono contenere caratteri "problematici" (come virgolette o newline) senza doverli scrivere in modo complicato. Si usano con il prefisso `R`:
    `R"(Questa è una stringa "grezza" che può contenere qualsiasi cosa!)"`
- *puntatore nullo*
  - Il valore `nullptr` rappresenta un puntatore nullo, cioè un riferimento "vuoto" a memoria.
  - Il tipo associato è `std::nullptr_t`.

6 - Riferimenti vs Puntatori

Un puntatore è un oggetto il cui valore (un indirizzo) si può riferire ad un altro oggetto, mentre un riferimento non è un oggetto vero e proprio, ma è una sorta di "nome alternativo" che consente di

accedere ad un oggetto esistente.
Inizializzazione:

- quando viene creato un riferimento, questo deve essere inizializzato, perche si deve sempre riferire ad un oggetto esistente, non esiste il concetto di riferimento nullo;
- un puntatore può non essere inizializzato (non consigliato), oppure essere inizializzato con `nullptr`, in questo caso non punterà ad alcun oggetto.
  Modificabilità:
- una volta creato un riferimento, questo si riferirà sempre allo stesso oggetto; non c'è modo di "riassegnare" un riferimento ad un oggetto differente;
- un puntatore (non qualificato `const`) può essere modificato per puntare ad oggetti diversi o a nessun oggetto.
  Accesso a oggetto riferito/puntato:
- ogni volta che si effettua una operazione su un riferimento, si sta implicitamente operando sull'oggetto riferito;
- nel caso dei puntatori abbiamo a che fare con due oggetti diversi: l'oggetto puntatore e l'oggetto puntato:
  - operazioni in lettura e scrittura applicate direttamente al puntatore accedono e, potenzialmente, modificano l'oggetto puntatore;
  - per accedere all'oggetto puntato occorre utilizzare la dereferenziazione.
  Qualificazione const:
- nel caso dei riferimenti si applica all'oggetto riferito;
- nel caso dei puntatori è possibile specificare il qualificatore const per ognuno dei due oggetti;
  - nella dichiarazione di un puntatore è possibile scrivere const due volte:
  - `const` a sinistra di `*`, si applica all'oggetto puntato;
  - `const` a destra di `*`, si applica all'oggetto puntatore.
  Lifetime:
  Dopo che il puntatore termina il suo ciclo di vita, solo il puntatore viene distrutto, non l'oggetto puntato. Analogo per i riferimenti.
  Nel caso in cui un riferimento viene inizializzato con un temporaneo, l'oggetto temporaneo finisce il ciclo di vita insieme al riferimento.
  Riferimenti "dangling"
  Come è possibile creare un dangling pointer (puntatore non nullo che contiene l'indirizzo di un oggetto non più esistente), è anche possibile creare una *dangling reference*.

I casi sopra considerati sono esclusivamente i riferimenti a lvalue (`T&`), la maggior parte delle osservazioni sono valide anche nel caso di riferimenti a rvalue (`T&&`).

7 - One Definition Rule

Un programma è suddiviso in più unità di traduzione (compilazione separata) che per interagire correttamente devono concordare su un interfaccia comune, per ridurre il rischio di inconsistenza dell'interfaccia si segue la regola **DRY** (Don't Repeat Yourself):
le dichiarazioni dell'interfaccia vengono scritte una volta sola, in uno o più header file
le unità di traduzione includono gli header file di cui hanno bisogno (senza ripeterle).
La **ODR** è una regola fondamentale che stabilisce come gestire correttamente definizioni e dichiarazioni all'interno di un programma. Serve a garantire che il comportamento del programma sia deterministico ed evita ambiguità o errori durante la compilazione e il linking.

> recall:
> **linker**: strumento che si occupa di collegare insieme tutti i file oggetto per creare il programma finale eseguibile.

ODR

La ODR stabilisce:

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 (in unità di traduzione diverse, come stabilito nel punto 1) a condizione che:
   4a. tali definizioni siano sintatticamente identiche;
   4b. tali definizioni siano semanticamente identiche.

Dettagli

Punto 1: definizione multipla di tipo in una unità di traduzione

Un'unità di traduzione (file sorgente .cc e tutti gli header file inclusi) può contenere una sola definizione di una data variabile, funzione, classe, ecc. Esempio di violazione:

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

Esempio corretto:

```
struct S { int a; }; //una sola definizione
extern S s; //dichiarazione della variabile
S s; //definizione della variabile
```

Punto 2: una definizione per variabili e funzioni globali

- caso banale: uso di variabile / funzione che è stata dichiarata ma non è mai stata definita, la compilazione in senso stretto va a buon fine ma verrà segnalato un errore al momento di generare il codice eseguibile
- definizioni multiple (a volte incosistenti) in traduzioni diverse:

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

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

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

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

Questo caso porta a errori di linking o comportamenti imprevedibili, perchè il linker potrebbe usare una definizione o l'altra (tra quella del file1 e quella del file2) senza avvisare.

Punto 3: definizioni di funzioni inline

Le funzioni inline devono essere definite in ogni unità di traduzione in cui vengono chiamate. Questo è necessario perchè il compilatore sostituisce la chiamata alla funzione con il corpo della funzione stessa durante la compilazione; infatti è per questo che conviene scriverne il corpo una sola volta in un header file.

Violazioni più avanzate

Punto 4: definizioni sintatticamente o semanticamente diverse

- *sintatticamente diversa*: due definizioni di una classe o tipo hanno una struttura diversa. Esempio:

```cpp
// file1.cc
struct S { int a; int b; };  // Definizione 1

// file2.cc
struct S { int b; int a; };  // Definizione 2 (ordine diverso)
```

- *semanticamente diversa*: anche se sintatticamente identiche, le definizioni hanno un significato diverso. Esempio:

```cpp
// file1.cc
typedef int T;
struct S { T a; T b; };

// file2.cc
typedef double T;  // Cambia il significato di T
struct S { T a; T b; };  // Sintassi identica, semantica diversa
```

Risolvere le violazioni della ODR

Per evitare errori legati alla ODR, si seguono queste pratiche:

1. **usare gli header file per dichiarazioni condivise**: tutte le dichiarazioni comuni (classi, funzioni, tipi) devono essere messe in un unico header file, incluso dove necessario.
2. **proteggere gli header file con guardie contro le inclusioni multiple**:
   Si utilizzano:
   - **guardie del preprocessore**:

```
#ifndef HEADER_NAME
#define HEADER_NAME
// Contenuto dell'header
#endif
```

esempio senza guardie:

```
// header.h
struct MyStruct {
    int a;
};

// file1.cpp
#include "header.h"

// file2.cpp
#include "header.h"

// main.cpp
#include "header.h"  // Incluso direttamente
#include "file1.cpp" // header.h viene incluso di nuovo!
#include "file2.cpp" // header.h viene incluso ancora una volta!
```

risultato: quando il compilatore processa il file main.cpp, troverà più definizioni di `MyStruct` e segnalerà un errore.

- oppure `#pragma once`: più semplice, ma non standard.

3. **non definire variabili o funzioni negli header file**: negli header file si mettono solo dichiarazioni, mentre le definizioni vanno nei file .cc .
4. **seguire la regola DRY**: evitare duplicazioni di definizioni, utilizzando gli stessi header file in tutte le unità di traduzione.

Costrutti ammessi e vietati negli header file

Ammessi:

- dichiarazioni di variabili, funzioni, classi, template;
- definizioni di funzioni inline o template;
- alias di tipi (`typedef` o `using`);
- namespace nominati.
  Vietati:
- definizioni di variabili e funzioni non-inline;
- namespace anonimi;
- `using namespace` (potrebbe creare conflitti di nomi).

Esempio pratico:

Consideriamo un programma che deve effettuare calcoli matematici e che necessita di usare:

- una classe per i numeri razionali;
- una classe per i polinomi a coefficienti razionali;

- una funzione che usa sia i razionali, sia i polinomi;
  Razionale.hh:

```
class Razionale {
        //definizione della classe
};
```

Polinomio.hh:

```
#include "Razionale.hh"
class Polinomio {
        //definizione della classe, usa Razionale
}
```

Calcolo.cc:

```
#include "Razionale.hh"
#include "Polinomio.hh" //violazione ODR punto 1
Razionale valuta(const Polinomio& p, const Razionale& r) {
        //calcola il valore di p in x
}
```

Analisi dell'errore: quando viene compilata l'unità di traduzione corrispondente al file sorgente `Calcolo.cc` si viola la ODR perché l'unità conterrà due definizione della classe `Razionale`:

- la prima dalla direttiva di inclusione;
- la seconda è ottenuta, indirettamente, dalla seconda direttiva di inclusione `#include "Polinomio.hh`.
  Se provassimo a modificare il file `Calcolo.cc`, per risolvere il problema, eliminando l'inclusione di `Razionale.hh`:
- viene a crearsi una *dipendenza indiretta* (nascosta);
- diminuirebbe la leggibilità del codice;
- verrebbe a complicarsi la sua manutenzione.
  La soluzione corretta sarebbe utilizzare le direttive del processore che impediscono di includere l'header file più di una volta, come `#pragma once`, oppure le guardie contro l'inclusione ripetuta.
  Quindi:
  Razionale.hh:

```
#ifndef RAZIONALE_HH
#define RAZIONALE_HH

class Razionale {
  // Definizione della classe
};

#endif
```

Polinomio.hh

```
#ifndef POLINOMIO_HH
#define POLINOMIO_HH

#include "Razionale.hh"

class Polinomio {
  // Definizione della classe, usa Razionale
};

#endif
```

Calcolo.cc

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

// Funzioni e codice che usano Polinomio e Razionale
```

8 - Passaggio di argomenti

Quando passiamo argomenti a una funzione, ci sono due modalità principali:

1. **passaggio per valore (`T`)**
   - si fa una copia dell'argomento e questa copia viene utilizzata all'interno della funzione;
   - **impatti**:
     - **pro**: la funzione lavora su una copia, quindi non modifica l'argomento originale.
     - **contro**: se l'argomento è un oggetto "grande" (ad esempio, un contenitore come un `std::vector`), la copia può essere costosa in termini di prestazioni.
   - è opportuno utilizzarlo per tipi semplici e piccoli, come int, float, o puntatori (che sono solo indirizzi di memoria).
2. **passaggio per riferimento (`const T&` o `T&`)**
   - la funzione lavora su un riferimento, cioè un alias dell'argomento originale. Non viene effettuata alcuna copia.
   - tipi di riferimento:
     - **riferimento a costante** (`const T&`): usato quando la funzione non deve modificare l'argomento originale;
     - **riferimento modificabile** (`T&`): usato quando la funzione deve modificare l'argomento originale.
   - **impatti**:
     - **pro**: evita la copia, quindi è più efficiente per oggetti grandi;
     - **contro**: con un riferimento modificabile, l'argomento originale può essere modificato, il che potrebbe essere indesiderato.
   - è opportuno utilizzare:
     - `const T&` per oggetti grandi che non devono essere modificati;
     - `T&` se la funzione deve modificare l'oggetto originale.

Ritorno dei valori dalle funzioni

Due modalità principali:

1. **ritorno per valore**:
    - caso più comune: restituisce una copia del valore;
    - preferibile rispetto alla seconda modalità perché:
        - le variabili locali della funzione vengono distrutte quando la funzione termina;
        - non si possono restituire riferimenti a queste variabili locali, altrimenti si otterrebbero riferimenti dangling.
2. **ritorno per riferimento**:
    - la funzione restituisce un riferimento a un oggetto esistente (non una copia);
    - è sicuro solo se l'oggetto a cui si restituisce il riferimento ha un ciclo di vita che supera quello della funzione;
    - **pro**: evita copie costose;
    - **contro**: deve essere usato con attenzione per evitare riferimenti dangling.

Riferimenti a rvalue

A partire da C++11, sono stati introdotti i riferimenti a rvalue (`T&&`), che permettono di lavorare con oggetti temporanei in modo efficiente. Questo concetto è utile soprattutto nella gestione delle risorse e soprattutto per sostituire operazioni di copia (costose) con operazioni di spostamento (più efficienti). L'uso esplicito di riferimenti a rvalue è raro, perchè gran parte di queste ottimizzazioni avvengono automaticamente.

Caso specifico del passaggio per valore: "passaggio per puntatore"

Quando passiamo un puntatore a una funzione, stiamo passando *per valore* l'indirizzo di memoria contenuto nel puntatore; tuttavia poiché un puntatore consente di accedere direttamente alla memoria, la funzione può modificare l'oggetto puntato. In ogni caso, è quasi sempre rimpiazzabile dal passaggio per riferimento. Un argomento di tipo puntatore ha 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

9 - Array e puntatori

Type decay

Quando si usa un'espressione che rappresenta un array viene applicato il **type decay**, ovvero l'array decade in un puntatore al suo primo elemento (tecnicamente, una trasformazione di lvalue). Questa conversione è utile per evitare copie costose degli array.
Gli array non possono essere copiati direttamente; passandoli a una funzione, viene passato per valore solo il puntatore al primo elemento.

```
void func (int* p) { /* ... */ }

int a[10];
func(a); // 'a' decade in un puntatore al primo elemento
```

L'indicizzazione degli array è, in realtà, un'abbreviazione che sfrutta l'aritmetica dei puntatori.

```
int a[100];
int b = 5;
int x = a[b]; //accedi all'elemento in posizione b dell'array
int y = *(a + b); //equivalente: somma b al puntatore 'a' e dereferenzia
```

- l'array 'a' decade in un puntatore al primo elemento (`&a[0]`);
- 'b' viene sommato al puntatore: ci si sposta in avanti di 'b' elementi nell'array;
- l'operatore `*` dereferenzia il puntatore per accedere al valore in memoria.
  Poiché la somma è commutativa, `*(a + b)` è equivalente a `*(b + a)` ; infatti `a[b]` è uguale a `b[a]` .
  Questo comportamento è valido solo per gli array 'classici'. Non funziona per altri tipi di contenitori, come `std::vector<T>` , perché non si basano su puntatori diretti.

Aritmetica dei puntatori

I puntatori supportano alcune operazioni aritmetiche. Se `ptr` è un puntatore che punta a un elemento di un array, possiamo:

1. **spostarsi in avanti** di n posizioni: `ptr + n` o `n + ptr` ;
2. **spostarsi indietro** di n posizioni: `ptr - n` .
   Queste operazioni funzionano a patto che il puntatore risultante:

- resti all'interno dell'array;
- oppure punti all'indirizzo immediatamente successivo alla fine dell'array (es. `a + 100` per un array di 100 elementi).

```
int a[100];
int* p1 = &a[10];  // Punta all'11° elemento
int* p2 = &a[5];   // Punta al 6° elemento

int distanza = p1 - p2; // distanza = 5
```

Iterazione con array e puntatori

Iterazione basata su indice

Il metodo più comune per iterare su un array è usare un indice:

```
int a[100];
for (int i = 0; i < 100; ++i) {
  // Fai qualcosa con a[i]
}
```

Iterazione basata su puntatore

Alternativa:

```
int a[100];
for (int* p = a; p != a + 100; ++p) {
```

```
  // Fai qualcosa con *p
}
```

Iterazione con coppie di puntatori

Se si conoscono due puntatori validi che delimitano un intervallo di un array (`p1` e `p2`), è possibile iterare senza conoscere esplicitamente l'inizio e la fine dell'array:

```
int* p1 = &a[10]; // Punta all'11° elemento
int* p2 = &a[20]; // Punta al 21° elemento

for (; p1 != p2; ++p1) {
  // Fai qualcosa con *p1
}
```

in questo caso si iterano solo gli elementi tra i due puntatori.
First e last
Quando si usa una coppia di puntatori per iterare, è comune chiamarli `first` (primo elemento incluso) e `last` (primo elemento escluso).
Se `first` e `last` coincidono, la sequenza è vuota e il ciclo non viene eseguito.

Iteratori generici

L'idioma basato su coppie di puntatori è stato generalizzato in c++ per funzionare con gli iteratori, che rappresentano una generalizzazione dei puntatori. Gli iteratori si usano con i contenitori della **Standard Template Library**, come `std::vector`, `std::list`, ecc.
Esempio con std::vector

```
#include <vector>

std::vector<int> v = {1, 2, 3, 4, 5};

//iterazione con iteratori
for (auto it = v.begin(); it != v.end(); ++it) {
        //fai qualcosa con *it
}
```

- `v.begin()` è un iteratore al primo elemento.
- `v.end()` è un iteratore alla posizione successiva all'ultimo elemento.

10 - lvalue e rvalue

In C++, le espressioni possono essere classificate in tre categorie principali, che aiutano a definire il *comportamento degli oggetti*, la *gestione della memoria* e l'*ottimizzazione*. Queste categorie sono:

lvalue (left value):

un'espressione che si riferisce a un oggetto con un'identità in memoria, quindi qualcosa che può essere "a sinistra" dell'operatore di assegnamento.

```
int i;
i = 5; // 'i' è un lvalue perché identifica un oggetto in memoria
```

xvalue (expiring value):

una forma speciale di lvalue che denota un oggetto "in scadenza", cioè che sta per terminare il proprio ciclo di vita. Ad esempio, il risultato di una chiamata a funzione che restituisce un oggetto temporaneo.

```
Matrix foo();
Matrix m = foo(); // 'foo()' è un xvalue: il valore temporaneo può essere spostato
```

prvalue (pure value):

un'espressione che rappresenta un valore puro, come una costante o il risultato di un'operazione, che non identifica alcun oggetto in memoria. Non è possibile prendere l'indirizzo di un prvalue o assegnargli un valore.

```
int x = 5; // '5' è un prvalue
int y = x + 1 // 'x + 1' è un prvalue
```

materializzazione dei prvalue:
in alcuni contesti, un prvalue può essere "materializzato", cioè il compilatore crea un oggetto temporaneo (un lvalue) che rappresenta il valore puro.

```
void foo(const int& x); // Accetta un riferimento costante a un lvalue.
foo(42);                // '42' (prvalue) viene materializzato come temporaneo.
```

Relazioni tra le categorie

- **glvalue (generalized value)**: l'unione tra lvalue e xvalue. Identifica sempre un oggetto in memoria.

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

  - un xvalue è un glvalue che denota un oggetto le cui risorse possono essere riutilizzate, tipicamente perchè sta terminando il suo lifetime;
  - un lvalue è un gvalue che non sia un xvalue;

  ```
  Matrix foo1() {
          Matrix m;
          //codice
          m.transpose(); //qui m è un lvalue (quindi glvalue)
          return m; //qui m è un xvalue (quindi glvalue)
  }
  /* '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 */
  ```

- **rvalue (right value)**: l'unione tra xvalue e prvalue. Indica un valore temporaneo o calcolato che non ha un'identità stabile in memoria.
  nota: gli xvalue sono sia glvalue, sia rvalue.

Riferimenti a lvalue e rvalue

C++ supporta due tipi principali di riferimenti, ciascuno associato a diverse categorie di espressioni:

1. **riferimenti a lvalue (`T&`)**:
   Questi riferimenti sono utilizzati per accedere a oggetti esistenti e non temporanei.

   ```
   int x = 10;
   int& ref = x; //riferimento a lvalue
   ```

2. **riferimenti a rvalue (`T&&`)**:
   questi riferimenti permettono di lavorare con oggetti temporanei (prvalue e xvalue). Sono fondamentali per implementare il "move semantics".

   ```
   Matrix foo();
   Matrix&& temp = foo(); // 'foo()' è un rvalue, catturato con un riferimento a rvalue
   ```

C++ 03: Costrutti e Assegnazioni

Ogni classe dispone di quattro funzioni speciali generate automaticamente:

1. **costruttore di default**: inizializza l'oggetto;
2. **costruttore di copia**: crea una copia di un altro oggetto della stessa classe;
3. **assegnazione per copia**: copia i valori da un altro oggetto;
4. **distruttore**: libera le risorse dell'oggetto.

```
struct Matrix {
    Matrix();                       // Costruttore di default
    ~Matrix();                      // Distruttore
    Matrix(const Matrix&);          // Costruttore di copia
    Matrix& operator=(const Matrix&); // Assegnazione per copia
};
```

Problemi:
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)
}
```

1. parametro passato per riferimento costante:
   `arg` è passato per riferimento a costante, il che significa che il chiamante non crea una copia completa dell'oggetto. Questo è utile per evitare il costo di copia quando il parametro è un oggetto grande; tuttavia non può essere modificato perché è `const`.
2. creazione di `res`:
   la variabile `res` viene inizializzata copiando l'oggetto `arg`, dunque avviene la *prima copia*, necessaria in quanto vogliamo modificare `res` senza influenzare `arg`.
3. ritorno di `res`:
   quando la funzione restituisce `res`, viene effettuata un'ulteriore copia per restituire l'oggetto al chiamante, dunque avviene la *seconda copia*.
   Inefficienze:
4. l'oggetto `arg` viene copiato per creare `res`. Questo è inefficiente quando il chiamante non ha più bisogno di `arg` e sarebbe disposto a lasciarlo modificare direttamente.

   ```
   Matrix m1;
   // Supponiamo che il chiamante non abbia più bisogno di m1
   Matrix m2 = bar(m1); // Il chiamante vuole che m1 venga usato per produrre m2
   ```

   anche se il chiamante non ha più bisogno di `m1`, la funzione `bar` non lo sa; quindi, per sicurezza, deve copiare `m1` (usando il costruttore di copia) per creare `res`. Questo è uno spreco di risorse perché:
   - la copia potrebbe essere costosa;
   - se il chiamante avesse un modo per segnalare che `m1` non è più necessario, la funzione potrebbe riutilizzare direttamente le risorse di `m1` invece di copiarle.
5. quando la funzione ritorna, il compilatore deve creare un'altra copia di `res` per trasferire il valore al chiamante.

   ```
   Matrix m2 = bar(m1);
   ```

   la funzione `bar` deve restituire un nuovo oggetto `Matrix`, questo significare che il valore di `res` deve essere copiato nel nuovo oggetto `m2`.
   Questo è un altro spreco di risorse perché:
   - l'oggetto `res` non è più necessario dopo il ritorno.
   - sarebbe più efficiente "spostare" le risorse interne di res direttamente nel risultato, evitando la copia.

Soluzione introdotta nel C++ 11: move semantic

Vengono aggiunte alle 4 funzioni speciali delle classi altre due che lavorano su riferimenti a rvalue:

- **costruttore per spostamento** (move constructor);
- **assegnamento per spostamento** (move assignment).

```
struct Matrix {
Matrix (); // default constructor
Matrix (const Matrix&); // copy constructor
Matrix& operator=(const Matrix&); // copy assignment
Matrix (Matrix&&); // move constructor
```

```
Matrix& operator=( Matrix&&); // move assignment
~Matrix(); // destructor
// altro
};
```

Con l'introduzione delle semantiche di spostamento (move semantics), i problemi di inefficienza sono stati risolti:

1. **costruttore di spostamento**:
   - se la classe `Matrix` implementa un costruttore per spostamento, il compilatore può evitare di copiare `res` quando la funzione ritorna; invece di fare una copia di `res`, il compilatore sposta le risorse interne di `res` nel risultato della funzione;
2. **nuova implementazione di bar**:
   con il *move constructor* la seconda copia non avviene più, quando la funzione ritorna, il compilatore riconosce che `res` è un xvalue e utilizza il costruttore per spostamento invece del costruttore di copia.

Per evitare la prima copia

Supponiamo che il chiamante si trovi a dover invocare la funzione bar con un lvalue `m`, ma non è interessato a preservarne il valore: quindi lo vorrebbe "spostare" nella funzione bar evitando la copia. Se si usa la chiamata `bar(m);` dato che `m` è un lvalue viene comunque invocata, almeno una volt, la copia. Per evitare questa copia inutile, occorre un modo per convertire il tipo di `m` da riferimento a lvalue (`Matrix&`) a riferimento a rvalue (`Matrix&&`):

```
Matrix m = bar(std::move(m1)); // Sposta m1 invece di copiarlo
```

la `std::move` non "muove" nulla, ma trasformando un lvalue in rvalue, lo rende "movable", lo spostamento avviene durante il passaggio del parametro.

Versione generale

Una versione ancora più generale combina entrambe le versioni di `bar`:

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

- Se il chiamante passa un **lvalue**, il costruttore di copia sarà usato per inizializzare `arg`.
- Se il chiamante passa un **rvalue**, il costruttore di spostamento sarà usato per inizializzare `arg`.
  Vantaggio: Una sola funzione gestisce sia gli lvalue sia gli rvalue in modo efficiente.

13 - Progettazione di un tipo di dato concreto

//Lez. 9-10-24
Consideriamo come esempio la classe Razionale, dando per scontato l'implementazione delle operazioni richiesta e ci concentriamo sul punto di vista del metodo.

Verranno sviluppate due unità di traduzione distinte, ognuna composta da due file sorgente (più header file di libreria)

- `Razionale.hh` : interfaccia
- `Razionale.cc` : implementazione
- `testRazionale.cc` : codice di test
  In modo intuitivo (non formale) useremo il TDD (Test Driven Design) che prevede la progettazione e lo sviluppo del codice sia guidata dai test che vengono scritti prima del codice dell'interfaccia e dell'implementazione.

testRazionale.cc

```
#include "RazionaleZaf.hh"

#include <iostream>

void do_something(Numerica::Razionale r) {
    (void) r;
}

void test01() {
    using Numerica::Razionale; //si trova in un solo namespace, non nel globale

    Razionale r; //costruttore  default

    Razionale r1(r); //costruttore di copia
    Razionale r2 = r; //costruttore di copia
    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 };

        //Razionale r9 = 1234; //costruzione implicita (da evitare!)
    //Razionale r9 = '?'

    do_something(r8);
    //do_something(1234); da evitare!!

    r = r1; //assegnamento per copia
    r = Razionale(1); //assegnamento per spostamento
    r2 = r1 = r; //concatenazione assegnamenti (da evitare)
}

void test02() {
    using Numerica::Razionale;

    Razionale r, r1, r2;

    //op aritmetici binari
    r = r1 + r2;
```

```
    r = r1 - r2;
    r = r1 * r2;
    r = r1 / r2; //divisione per zero?

    //Operatori aritmetici unari
    r = -r1;
    r = +r1;

    //op assignement
    r += r1;
    r -= r1;
    r *= r1;
    r /= r1;

    //concatenazione assegnamenti
    r = (r += r1);
    r += r += r1 -=r1; //anche no?
}

void test03 () {
    using Numerica::Razionale;
    Razionale r;

    //pre incremento e decremento (prima fa l'incremento e poi restituisce il nuovo
valore)
    ++r;
    --r;
    ++++r;

//post incremento e decremento (ha una semantica meno intuitiva, incrementa ma
restituisce il vecchio risultato (+ per restituire il vecchio valore va a creare una
copia del numero ->costo elevato)) (non andrebbe mai usato)
    r++;
    r--;

    +++++++++r; //anche no?
}

void test04() {
    using Numerica::Razionale;

    Razionale r, r1, r2;

    r == r1;
    r != r2;
    r < r2;
    r > r2;
    r <= r1;
    r >= r2;
}

void test05() {
    using Numerica::Razionale;

    Razionale r1, r2;
```

```
    //r1 = r1 + 1234 //aritmetica mista?
    //r1 = 1234 + r1 //aritmetica mista?

    //gli op di stream non possono essere membri della classe

    std::cin >> r1;
    std::cin >> r1 >> r2;

    std::cout << r1;
    //std:: r1 << "+" << r2 << "=" << (r1 + r2) << std::endl;
}

void test06(const Numerica::Razionale& r) {
    std::cout << r; //se in 'std::ostream& operator<<(std::ostream& os, const
Razionale& x);' andiamo a togliere const non compila

}


int main () {
    test01();
    test02();
    test03();
    test04();
    test05();
}

/*
const PRO: il valore di un nostro risultato non potrà mai essere modificato
inavvertitamentee -> aiuta nel
           debugging
           const correctness
*/
```

Razionale.hh

```
#pragma once
#include <iostream>

namespace Numerica {
class Razionale{
public:

    using Intero = long; //se dovessi cambiarlo un giorno lo dovrò modificare solo in
questo punto

    Razionale() = default;

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

    //spostamento
```

```
    Razionale(Razionale&&) = default;
    Razionale& operator=(Razionale&&) = default;

    ~Razionale() = default;

    explicit Razionale(const Intero& num, const Intero& den = 1); //non lo
implemento, lo dichiaro solamente perche siamo nell'interfaccia
                                                        // 'den = 1' quando il
denominatore non è fornito viene messo 1 -> r9 compila -> bisogna disabilitare la
conversione implicita con la keyword 'explicit'
    //lo mettiamo fuori: Razionale operator+(const Razionale& y) const; //possiamo
dichiararlo all'interno della classe
    Razionale operator+() const;
    Razionale operator-() const; //restituiscono un nuovo razionale, non vanno a
modificarre quello originale
    void nega();
    Razionale& operator+=(const Razionale& y); //la funzione non è const perchè deve
andare a modificare il razionale sul quale è invocato e lo ritorno per riferimento
(per avere la possibilità di concatenare: es. '(r += r1) += r2', se non vogliamo
possiamo cambiare da Razionale& -> void), se lo vogliamo solo in lettura utilizziamo:
'const Razionale&'
    Razionale& operator-=(const Razionale& y);
    Razionale& operator*=(const Razionale& y);
    Razionale& operator/=(const Razionale& y);

    Razionale& operator++(); //Razionale& perche di default vogliamo che sia
concatenabile es. '++ ++r'
    Razionale& operator--();

    bool operator==(const Razionale& y) const;
    bool operator<(const Razionale& y) const;

    void print_to (std::ostream& os) const; //lo utilizziamo perche l'overloading di
<< e >> hanno bisogno di accedere ai campi privati della classe
    void read_from (std::istream& is);
private:
    Intero num_;
    Intero den_;

    void semplifica();
}; //class razionale

Razionale operator+ (const Razionale& x, const Razionale& y); //interfaccia uguale,
cambia a livello di implementazione (essendo fuori posso accedere solo alle parti
pubbliche della classe e non a quelle private)
Razionale operator- (const Razionale& x, const Razionale& y);
Razionale operator* (const Razionale& x, const Razionale& y);
Razionale operator/ (const Razionale& x, const Razionale& y);


inline bool operator!=(const Razionale& x, const Razionale& y) {
    return !(x == y);
}

inline bool operator>(const Razionale& x, const Razionale& y) {
```

```
    return y < x;
}

bool operator<=(const Razionale& x, const Razionale& y);
bool operator>=(const Razionale& x, const Razionale& y);

std::ostream& operator<<(std::ostream& os, const Razionale& x);
std::ostream& operator>>(std::ostream& is, Razionale& x);

} //namespace Numerica
```

Razionale.cc

```
#include <iostream>
#include <numeric>
#include <cassert>

#include "../../include/Razionale/Razionale.hh"

namespace Numerica {

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

        void Razionale::semplifica() {
                assert(den_ > 0);
                Intero gcd = std::gcd(num_, den_);
        if (gcd != 1) {
            num_ /= gcd;
            den_ /= gcd;
        }
        }

        Razionale::Razionale() : Razionale(0, 1) {
                assert(check_inv());
        }

        Razionale::Razionale(const Intero& num, const Intero& den) :
                num_{num}, den_{den} {

                assert(den != 0);

                if (num_ == 0) {
            den_ = 1;
            assert(check_inv());
            return;
```

```
}

        if (den < 0) {
                den_ = -den;
                num_ = -num;
        }

        semplifica();
        assert(check_inv());
}

Razionale::Razionale(const Razionale& other) :
        num_{other.num_}, den_{other.den_} {

        assert(other.check_inv() && check_inv());
}

const Razionale& Razionale::operator=(const Razionale& other) {
        num_ = other.num_;
        den_ = other.den_;

        assert(other.check_inv() && check_inv());

        return *this;
}

Razionale::Razionale(Razionale&& other) :
        num_{std::move(other.num_)},
        den_{std::move(other.den_)} {

        std::cout << "Costruttore per spostamento: " << *this << std::endl;
        assert(check_inv());
}

const Razionale& Razionale::operator=(Razionale&& other) {
        num_ = std::move(other.num_);
        den_ = std::move(other.den_);

        std::cout << "Assegnamento per spostamento:" << *this << std::endl;
        assert(check_inv());
        return *this;
}

Razionale& Razionale::operator+=(const Razionale& other) {
        assert(other.check_inv() && check_inv());

        num_ = (this->num_  * other.den_) + (this->den_ * other.num_);
        den_ *= other.den_;

        semplifica();
        assert(check_inv());

        return *this;
}
```

```
Razionale& Razionale::operator-=(const Razionale& other) {
	assert(other.check_inv() && check_inv());

	num_ = (this->num_  * other.den_) - (this->den_ * other.num_);
	den_ *= other.den_;

	semplifica();
	assert(check_inv());

	return *this;
}

Razionale& Razionale::operator*=(const Razionale& other) {
	assert(other.check_inv() && check_inv());

	num_ *= other.num_;
	den_ *= other.den_;

	semplifica();
	assert(check_inv());

	return *this;
}

Razionale& Razionale::operator/=(const Razionale& other) {
	assert(other.check_inv() && check_inv());
	assert(other.num_ != 0);

	num_ *= other.den_;
	den_ *= other.num_;

	if (this->den_ < 0) {
		this->den_ = -this->den_;
		this->den_ = -this->den_;
	}

	this->semplifica();
	assert(check_inv());

	return *this;
}

bool Razionale::operator==(const Razionale& other) const {
	assert(other.check_inv() && check_inv());

	return (
		num_ == other.den_ &&
		den_ == other.den_
	);
}

bool Razionale::operator!=(const Razionale& other) const {
	assert(other.check_inv() && check_inv());

	return !(*this == other);
```

```
}

bool Razionale::operator<(const Razionale& other) const {
        assert(other.check_inv() && check_inv());

        return (
                (num_ * other.den_) <
                (other.num_ * den_)
        );
}

bool Razionale::operator<=(const Razionale& other) const {
        assert(other.check_inv() && check_inv());

        return (
                (num_ * other.den_) <=
                (other.num_ * den_)
        );
}

bool Razionale::operator>(const Razionale& other) const {
        assert(other.check_inv() && check_inv());

        return (
                (num_ * other.den_) >
                (other.num_ * den_)
        );
}

bool Razionale::operator>=(const Razionale& other) const {
        assert(other.check_inv() && check_inv());

        return (
                (num_ * other.den_) >=
                (other.num_ * den_)
        );
}

Razionale Razionale::operator+() const {
        assert(check_inv());

        Razionale res(*this);

        assert(res.check_inv());
        return res;
}

Razionale Razionale::operator-() const {
        assert(check_inv());

        Razionale res(*this);
        res.num_ = -res.num_;

        assert(res.check_inv());
        return res;
```

```
}

Razionale& Razionale::operator++() {
        assert(check_inv());
        this->num_ += this->den_;
        assert(check_inv());

        return *this;
}

Razionale Razionale::operator++(int t) {
        assert(check_inv());
        Razionale res(*this);
        assert(res.check_inv());

        ++(*this);

        assert(check_inv());
        return res;
}

Razionale& Razionale::operator--() {
        assert(check_inv());
        this->num_ -= this->den_;
        assert(check_inv());

        return *this;
}

Razionale Razionale::operator--(int t) {
        assert(check_inv());
        Razionale res(*this);
        assert(res.check_inv());

        --(*this);

        assert(check_inv());
        return res;
}

const Razionale::Intero& Razionale::num() const {
        return num_;
}

const Razionale::Intero& Razionale::den() const {
        return den_;
}

void Razionale::print(std::ostream& os) const {
        assert(check_inv());
        os << num_;

        if (den_ != 1) {
                os << "/" << den_;
        }
```

```
	}

	std::ostream& operator<<(std::ostream& os, const Razionale& r) {
		r.print(os);
		return os;
	}

	std::istream& operator>>(std::istream& is, Razionale& r) {
		char slash;
		Razionale::Intero num, den;

		is >> num >> slash;

		if (isspace(slash)) {
			r = Razionale(num, 1);
		} else {
			assert(slash == '/');
			is >> den;
			r = Razionale(num, den);
		}

		return is;
	}

	Razionale operator+(const Razionale& left, const Razionale& right) {
		Razionale res(left);
		res += right;
		return res;
	}

	Razionale operator-(const Razionale& left, const Razionale& right) {
		Razionale res(left);
		res -= right;
		return res;
	}

	Razionale operator*(const Razionale& left, const Razionale& right) {
		Razionale res(left);
		res *= right;
		return res;
	}

	Razionale operator/(const Razionale& left, const Razionale& right) {
		Razionale res(left);
		res /= right;
		return res;
	}

} //! namespace Numerica
```

Invariante di classe

Proprietà che deve essere soddisfatta dalla rappresentazione scelta per il tipo di dati: quando possibile, è utile codificarne il controllo in un metodo della classe (`check_inv`) e verificarne la validità mediante le

asserzioni. Un oggetto che non viola l'invariante di classe è ben formato. Deve essere garantito che le operazioni della classe rendano vera l'invariante quando vengono creati nuovi oggetti e che la mantengano valida quando oggetti già esistenti sono modificati.

15 - Programmazione per contratto

La programmazione per contratto (*Design by contract*) è un paradigma in cui una classe o una funzione viene definita tramite un contratto tra il programmatore che la implementa e quello che la utilizza.
Questo contratto specifica chiaramente:

1. **precondizioni**: le condizioni che devono essere vere prima che una funzionalità venga invocata, responsabilità dell'utente garantire che siano soddisfatte;
2. **postcondizioni**: le condizioni che devono essere vere dopo l'esecuzione della funzionalità, responsabilità dell'implementatore garantire che vengano rispettate;
3. **invarianti di classe**: proprietà che devono essere sempre vere per gli oggetti della classe, sia prima che dopo l'esecuzione di qualsiasi funzionalità.
   Questa struttura permette di rendere il comportamento di una classe prevedibile e robusto, poichè ciascuna delle parti (implementatore e utilizzatore) sa esattamente quali sono i propri obblighi e cosa aspettarsi.

Forma del contratto

Un contratto si può rappresentare come una implicazione logica:
precondizioni ⇒ postcondizioni
Questo significa che se le precondizioni sono soddisfatte, allora l'implementatore si impegna a garantire le postcondizioni. Se invece le precondizioni non sono valide, l'implementatore non ha alcun obbligo.
Spesso si includono esplicitamente anche le invarianti di classe, specificando il contratto:
precondizioni AND invarianti ⇒ postcondizioni AND invarianti
In questo caso:

- le precondizioni e postcondizioni sono definite "al netto" delle invarianti, cioè assumono che le invarianti siano già rispettate.

Esempio classe Razionale

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  assert(x.check_inv() && y.check_inv()); // Controllo delle invarianti di ingresso
  assert(y != Razionale(0));              // Precondizione: y deve essere diverso da 0

  Razionale res = x;
  res /= y;

  // Invarianti in uscita (non serve verificarle per x e y, essendo const)
  assert(res.check_inv());
  return res;
}
```

- **precondizione**: `y` deve essere diverso da 0;
- **invarianti**: gli oggetti `x` e `y` devono rispettare le loro condizioni di validità interna;
- **postcondizione**: il risultato restituito deve essere corretto e mantenere le invarianti.

Contratti Narrow e Wide

Contratto Narrow (stretto)

Un contratto narrow specifica che l'implementatore fornisce la funzionalità solo se vengono rispettate determinate precondizioni. L'onere di garantire che queste siano vere è totalmente a carico dell'utilizzatore.
Caratteristiche:

- l'implementatore non deve gestire scenari in cui le precondizioni non sono valide;
- le precondizioni vengono spesso imprementate tramite asserzioni (`assert`), che funzionano solo in modalità di debug;
- più efficiente, poichè non richiede controlli aggiuntivi durante l'esecuzione.

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  assert(y != Razionale(0)); // L'utilizzatore deve garantire che y ≠ 0
  Razionale res = x;
  res /= y;
  return res;
}
```

Ad esempio:

- quando si accede ad un elemento di un array, l'onere di controllare la validità dell'indice è a carico del programmatore;
- a livello di libreria standard spetta all'utente controllare che `std::vector<T>` non sia vuoto prima di eliminare l'ultimo elemento con la `pop_back()` .

Contratto wide (ampio)

Un contratto wide prevede che l'implementatore si occupi di verificare le leggittimità delle precondizioni e gestisca eventuali violazioni. Questo sposta parte dell'onere dell'utilizzatore all'implementatore.
Caratteristiche:

- l'implementatore deve gestire tutte le situazioni, inclusi casi non validi;
- i controlli devono essere espliciti e attivi anche in modalità di produzione (quindi non è possibile utilizzare le asserzioni);
- è meno efficiente e più complesso da implementare, ma riduce gli errori lato utente.

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  if (y == Razionale(0)) {
    throw DivByZero(); // L'implementatore gestisce il caso di errore
  }
  Razionale res = x;
  res /= y;
  return res;
}
```

Differenze principali

contratto Narrow	contratto Wide
l'utente deve garantire le precondizioni	l'implementatore verifica le precondizioni
più efficiente	meno efficiente
controlli con `assert` (solo in debug)	controlli espliciti, sempre attivi.

Contratti nel linguaggio C++ e nelle libreria standard

Lo standard del C++ descrive ogni funzionalità del linguaggio e della libreria con un contratto implicito o esplicito, classificando il comportamento in base al livello di specifica:

Categorie di comportamento

1. **comportamento specificato (specified behavior)**:
   - il comportamento è definito esattamente dallo standard e ogni implementazione deve rispettarlo;
   - es: l'operazione di somma tra interi produce sempre il risultato atteso se non si verificano overflow.
2. **comportamento definito dall'implementazione (implementation-defined behavior)**:
   - lo standard permette all'implementazione di scegliere una modalità, ma la scelta deve essere documentata;
   - es: la dimensione dei tipi interi (`int` , `long` ...) dipende dall'implementazione ma deve essere dichiarata.
3. **comportamento non specificato (unspecified behavior)**:
   - lo standard non specifica quale comportamento debba essere adottato e l'implementazione non è tenuta a documentarlo;
   - es: l'ordine di valutazione degli argomenti in una chiamata di funzione.
4. **comportamento non definito (undefined bahavior)**:
   - si verifica quando vengono violate le precondizioni. Lo standard non specifica alcun vincolo e l'implementazione può comportarsi in modo arbitrario;
   - es: accedere a un elemento di un array fuori dai limiti, scrivere su una variabile `const` , overflow su tipi interi con segno;
   - pericoloso perchè può generare risultati imprevedibili, tra cui crash o modifiche indesiderate.
5. **comportamento locale (locale-specific behavior**:
   - dipende dalle impostazioni locali (lingua, cultura, convenzioni);
   - es: l'interpretazione dei caratteri dipende dal "character set" locale.

es. 3 prova in itinere 2016
La seguente classe presenta alcuni problemi che ne rendono l'utilizzo problematico. Individuare i problemi ed indicare una possibile soluzione (riscrivendo l'interfaccia).

```
struct Matrix {
        //...
        size_type num_rows();
        size_type num_cols();
        value_type& get (size_type row, size_type col);
        Matrix& operator-();
```

```
        Matrix& operator+=(Matrix y);
        Matrix& operator+(Matrix y);
        void print(ostream os);
        //...
};
```

correzione:

```
Struct Matrix {
        size_type num_rows() const;
        size_type num_cols() const;
        value_type& get (size_type row, size_type col); //ha senso fornire un metodo
in scrittura
        const value_type& get(size_type row, size_type col) const;
        Matrix operator-() const; //calcola un nuovo valore che è il negato della
mat, perciò restituisco per valore
        Matrix& operator+=(const Matrix& y);
        Matrix operator+(const Matrix& y) const; //calcola un nuovo valore perciò va
restituito
        void print(ostream& os) const; //const a destra per poter passare anche le
mat marcate const,
}
```

codice utente:

```
void foo(Matrix& m) {
        m.get(1,1) = 17;
        Matrix y;
        m += y; // m = m + y => ha senso tenere y const
        Matrix z;
        z = m + y;
}

void bar(const Matrix& m) {
        std::cout << m.get(1,1); //non è possibile utilizzare la terza funzione
perchè non è marcata const
}
```

11 - Risoluzione dell'overloading

L'overloading in C++ è una caratteristica che permette di definire più funzioni con lo stesso nome, distinguendole in base al numero e/o al tipo dei loro parametri, particolarmente utile per mantenere leggibile e coerente il codice quando si implementano funzionalità simili per tipi diversi.
cpp
float sqrt(float arg);
double sqrt(double arg);
long sqrt(long arg);

Il compilatore si occupa di scegliere automaticamente quale versione invocare in base al tipo dell'argomento.

Anche gli operatori supportano l'overloading, il che rende possibile estendere questi meccanismi anche ai tipi definiti dall'utente.

Risoluzione dell'overloading

La risoluzione di una chiamata a funzione sovraccaricata avviene *staticamente*, ovvero a tempo di compilazione. Il compilatore segue un processo ben definito per determinare quale funzione chiamare:

1. *individuazione delle funzioni candidate*: si selezionano tutte le funzioni con il nome appropriato e visibili nel punto di chiamata;
2. *selezione delle funzioni utilizzabili*: si verifica se, tra le funzioni candidate, esistono quelle i cui parametri possono accettare gli argomenti della chiamata, eventualmente applicando conversioni implicite;
3. *scelta della migliore funzione utilizzabile*: si confrontano le funzioni utilizzabili per stabilire quale richiede il minor numero di conversioni o quelle meno invasive (come promozioni rispetto a conversioni standard o definite dall'utente).

Fase 1: Funzioni candidate

L'insieme delle funzioni candidate è costituito dalle funzioni dichiarate all'interno dell'unità di traduzione che soddisfano questi requisiti:

- **nome corrispondente**: la funzione deve avere lo stesso nome della funzione chiamata.
  - tenere presente che per gli operatori la sintassi della chiamata di funzione può variare:
    - *sintassi operatore*:
      - operatore prefisso `- -r`
      - operatore infisso `r1 + r2`
      - operatore postfisso `vect[5]`
    - *sintassi funzionale*:
      - `r.operator++()`
      - `operator+(r1, r2)`
      - `vect.operator[](5)`
- **visibilità**: la funzione deve essere visibile nel punto della chiamata.
  - *chiamate a metodi di una classe*: se un metodo è chiamato tramite `obj.metodo()` (o `ptr->metodo()`), la ricerca avviene nello scope della classe del tipo **statico** dell'oggetto/puntatore.

    ```
    struct S { void foo(); };
    struct T : public S { void foo(int); };

    S* ptr = new T;
    ptr->foo(); // Si cerca in S, non in T.
    ```

  - *funzioni con qualificatore*: se una funzione viene chiamata usando un qualificatore (`namespace::funzione`), la ricerca inizia nello scope del namespace qualificato.

    ```
    namespace N { void foo(int); }
    void foo(char);
    ```

```
int main() {
    N::foo(42); // La funzione globale foo(char) non è visibile.
}
```

- *hiding (nascosta dall'overloading)*: una funzione dichiarata in una classe derivata può nascondere una funzione con lo stesso nome nella classe base,

```
struct S { void foo(int); };
struct T : public S { void foo(char); };

T t;
t.foo(5); // Errore: la funzione S::foo(int) è nascosta.
```

a meno che non venga esposta esplicitamento con `using`

```
struct T : public S {
    using S::foo;
    void foo(char);
};

T t;
t.foo(5); // Funziona: le due funzioni vanno in overloading.
```

- **Argument Dependent Lookup (ADL)**: la regola ADL amplia la ricerca delle funzioni candidate, includendo i namespace associati ai tipi degli argomenti. Quindi
  - se la chiamata di funzione non è qualificata;
  - se vi sono (uno o più) argomenti di un tipo definito dall'utente (cioè hanno tipo struct/classe/enum S, o riferimento a S o puntatore a S, possibilmente qualificati);
  - se il tipo suddetto è definito nel namespace N
    ⇒ allora la ricerca delle candidate viene effettuata anche all'interno del namespace N.

```
namespace N {
    struct S {};
    void foo(S);
}

int main() {
    N::S s;
    foo(s); // Trova automaticamente N::foo grazie all'ADL.
}
```

Fase 2: Funzioni utilizzabili

Una funzione candidata è **utilizzabile** se soddisfa entrambe queste condizioni:

- **numero degli argomenti**: il numero degli argomenti nella chiamata deve corrispondere a quello dei parametri della funzione, considerando anche:
  - parametri con valori di default;
  - l'argomento implicito `this` nei metodi non statici.

- **compatibilità dei tipi**: ogni argomento della chiamata deve poter essere convertito nel tipo del corrispondente parametro della funzione. Le conversioni implicite ammesse includono:
  - **conversioni esatte** (identità, qualificazioni, trasformazioni di lvalue);
  - **promozioni** (es. da `int` a `long`);
  - **conversioni standard** (es. da `float` a `double`);
  - **conversioni definite dall'utente** (operatori di conversioni o costruttori).

Fase 3: Funzione migliore

Se esistono più funzioni utilizzabili, il compilatore seleziona quella che richiede le conversioni meno invasive. Le regole per determinare la "migliore" funzione si basano sul *rank* delle conversioni richieste:

1. **conversioni esatte** (identità) sono preferite su tutto;
2. **promozioni** sono preferite sulle conversioni standard;
3. **conversioni standard** sono preferite alle conversioni definite dall'utente.
   Se due funzioni sono equivalenti per tutti gli argomenti, il compilatore genera un errore di ambiguità.

```
void foo(int);
void foo(double);

foo(42); // Match perfetto su foo(int), nessuna ambiguità.
foo(42.0); // Match perfetto su foo(double), nessuna ambiguità.
foo('a'); // Ambiguità: 'a' può essere promosso a int o convertito a double.
```

Osservazioni

- le regole viste non prendono in considerazione la presenza di funzioni candidate ottenute istanziando template di funzione;
- quando si scelgono le funzioni *candidate*, il numero e il tipo dei parametri non sono considerati;
- nel caso di invocazione di un metodo di una classe, il fatto che il metodo sia dichiarato con accesso `public`, `private` o `protected` non ha nessun impatto sul processo di risoluzione dell'overloading; infatti la miglior funzione utilizzabile ma non accessibile non viene sostituita da un'altra funzione utilizzabile e accessibile.

Esercizio

1. Mostrare il processo di risoluzione dell’overloading per le seguenti chiamate di funzione. Per ogni chiamata, indicare l’insieme delle funzioni candidate, l’insieme delle funzioni utilizzabili e, se esiste, la migliore funzione utilizzabile.

```
namespace NB {
        class D{};
} //namespace NB

namespace NA {
        class C{};

        void f(int i); //funzione #1
        void f(double d, C c = C()); //funzione #2
```

```
        void g(C c = C(), NB::D d = NB::D()); //funzione #3

        void h(C c); //funzione #4

        void test1() {
                f(2.0); //chiamata A
        }
} //namespace NA

namespace NB {
        void f(double d); //funzione #5

        void g(NA::C c = NA::C(), D d = D()); //funzione #6

        void h(NA::C c, D d); //funzione #7

        void test2(double d, NA::C c) {
                f(d); //chiamata B
                g(c); //chiamata C
                h(c); //chiamata D
        }
} //namespace NB

void f(NA::C c, NB::D d); //funzione #8

void test3(NA::C c, NB::D d) {
        f(1.0); //chiamata E

        g(); //chiamata F
        g(c); //chiamata G
        g(c,d); //chiamata H
}
```

CHIAMATA A:

- fun. candidate:
  - #1
  - #2
    //le altre funzioni con nome `f` non sono visibili
- fun. utilizzabili:
  - #1
  - #2
- fun. migliore:
  - #2
    CHIAMATA B:
- fun. candidate:
  - #5
- fun. utilizzabili:
  - #5

- fun. migliore:
  - #5
  CHIAMATA C:
- fun. candidate:
  - #6
  - #3
- fun. utilizzabili:
  - #6 //tecnicamente non è un match perfetto, perchè la chiamata passa un lvalue mentre la funzione richiede un rvalue (il parametro della funzione viene passato per valore non per riferimento)
  - #3
- fun. migliore:
  non esiste ⇒ ambiguità
  CHIAMATA D:
- fun. candidate:
  - #7
  - #4
- fun. utilizzabili:
  - #4
- fun. migliore:
  - #4
  CHIAMATA E: //non si applica la ADL perchè non contiene nessun tipo definito dall'utente
- fun. candidate:
  - #8
- fun. utilizzabili:
  non ha funzioni utilizzabili
- fun. migliore:
  non esiste
  CHIAMATA F: //le uniche funzioni g esistenti sono all'interno di namespace
- fun. candidate: non esiste
- fun. utilizzabili: non esiste
- fun. migliore: non esiste
  CHIAMATA G:
- fun. candidate:
  - #3
- fun. utilizzabili:
  - #3
- fun. migliore:
  - #3
  CHIAMATA H:
- fun. candidate:
  - #3
  - #6
- fun. utilizzabili:
  - #3

    - #6
- fun. migliore:
  non esiste ⇒ ambiguità

2. Mostrare il processo di risoluzione dell'overloading per le seguenti chiamate di funzione. Per ogni chiamata, indicare: l'insieme delle funzioni candidate; l'insieme delle funzioni utilizzabili; la migliore funzione utilizzabile (se esiste); il motivo di eventuali errori di compilazione.

```
#include <string>

namespace N {
        class C {
        private:
                std::string& first(); //funzione #1

        public:
                const std::string& first() const; //funzione #2

                std::string& last(); //funzione #3
                const std::string& last() const; //funzione #4

                C(const char*); //funzione #5
        }; //class C

        void print(const C&); //funzione #6
        std::string& f(int); //funzione #7

} //namespace N

class A {
public:
        explicit A(std::string&); //funzione #8
}; //class A

void print(const A&); //funzione #9

void f(N::C& c) //funzione #10
{
        const std::string& f1 = c.first(); //chiamata A
        std:string& f2 = c.first(); //chiamata B
        const std::string& l1 = c.last(); //chiamata C
        std::string& l2 = c.last(); //chiamata D
}

void f(const N::C& c) //funzione #11
{
        const std::string& f1 = c.first(); //chiamata E
        std::string& f2 = c.first(); //chiamata F
        const std::string& l1 = c.last(); //chiamata G
        std::string& l2 = c.last(); //chiamata H
}

int main() {
        N::C c("begin"); //chiamata I
```

```
        f(c); //chiamata L
        f("middle"); //chiamata M
        print("end"); //chiamata N
}
```

	f. candidate	f. utilizzabili	f. migliore	motivo:
A	#1, #2	none	none	`c.first()` invocherebbe la funzione #1, però è privata quindi non può essere utilizzata. Invocherebbe la #2, però `c` non è costante, quindi non è possibile invocarla
B	#1, #2	none	none	idem chiamata A
C	#3, #4	#3	#3	viene chiamata la #3 perché anche se `l1` è un riferimento costante ad una stringa, `c` non lo è
D	#3, #4	#3	#3	
E	#1, #2	#2	#2	`c` è un riferimento const, quindi è possibile chiamare la #2 marcata const
F	#1, #2	none	none	ragionamento simile alla E, viene restituito un riferimento const a stringa che non può essere convertito in riferimento non const
G	#3, #4	#4	#4	
H	#3, #4	none	none	stesso motivo della F
I	#5	#5	#5	
L	#10, #11, #7	#10, #11	#10	chiamiamo la #10 perchè la `c` creata nella chiamata I non è const
M	#10, #11	#10, #11	ambiguità	non è presente nessuna funzione utilizzabile, perchè le chiamate aspettano come argomento un oggetto di tipo `C`
N	#9	none	none	non è presente nessuna funzione utilizzabile, perchè le chiamate aspettano come argomento un oggetto di tipo `A`

12 - Conversioni implicite

La conversioni implicite di tipo avvengono automaticamente durante l'esecuzioni di un programma, queste conversioni consentono al compilatore di rendere compatibili tipi diversi di dati durante assegnazioni, passaggi di parametri o operazioni. Sono suddivise in 4 categorie principali:

1 - le corrispondenze "esatte":

sono quelle conversioni implicite che preservano il valore dell'argomento e si suddividono in tre sottocategorie:

1a. Identità (match perfetti):

Tecnicamente non è una conversione, ma viene considerata un caso speciale, si verifica quando il tipo del parametro e il tipo dell'argomento coincidono esattamente.
esempi: date le dichiarazioni `int i; const int& r = i`

argomento	tipo argomento	tipo parametro
5	int	int
i	int&	int&
&i	int*	int*
r	const int&	const int&
5.2	double	double

1b. Trasformazioni di lvalue:

Le trasformazioni di lvalue riguardano cambiamenti specifici che un lvalue subisce per adattarsi al tipo richiesto.
Tipi di trasformazioni:

1. **da lvalue a rvalue**: un lvalue viene utilizzato come un valore (es. passaggio per valore), si verifica quando siamo interessati al valore memorizzato in una locazione e non alla locazione in quanto tale;
2. **decay array-puntatore**: un array viene trattato come un puntatore al suo primo elemento;
3. **decay funzione-puntatore**: una funzione viene convertita in un puntatore alla funzione stessa;
   Esempi: date le dichiarazioni `int i; int a[10]; void foo();`

argomento	tipo argomento	tipo parametro	
i	int&	int	da lvalue a rvalue
a	int[10]	int*	decay array → puntatore
foo	void(int)	void(*)(int)	decay funzione → puntatore

1c. Conversioni di qualificazione:

Avvengono quando viene aggiunto il qualificatore `const` a un riferimento o a un puntatore. Questo accade per garantire la sicurezza del tipo, impedendo modifiche non intenzionali ai dati originali.
Esempi: data la dichiarazioni `int i`

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

2 - Promozioni

Sono un tipo speciale di conversione implicita che preserva il valore dell'argomento. Sono progettate per adattare i tipi più piccoli o meno precisi a tipi più grani o più precisi a tipi più grandi o più precisi.

2a. Promozioni intere:

i tipi interi piccoli, come `char` e `short`, non sono direttamente rappresentabili nei registri del processore. Per questo motivo, vengono promossi a `int` o `unsigned int` prima di eseguire

operazioni.
Esempi:

- `char` o `short` → `int` (o `unsigned int`)
- `bool` → `int`

2b. Promozioni floating-point:

I numeri in virgola mobile rappresentati come `float` vengono promossi a `double` per garantire maggiore precisione durante i calcoli.

2c. Promozioni delle costanti di enumerazione

Le costanti di enumerazione definite in C++2003 possono essere promosse al più piccolo tipo intero (almeno `int`) che possa contenerle.
Esempio:

```
enum Color { Red = 1, Green = 2, Blue = 3};
```

- `Red` viene promosso a `int`.

3 - Conversioni Standard

Includono tutte le altre conversioni implicite che non rientrano nelle categorie precedenti. Non garantiscono sempre la preservazione del valore e comprendono conversioni tra tipi numerici, tra puntatori e tra riferimenti.
Esempi

- da `int` a `long` (non è una promozione);
- da `char` a `double`;
- da `double` a `int`.

Conversioni tra puntatori e riferimenti

1. **Costante intera 0 e `nullptr`**
   - la costante 0 può essere convertita a un puntatore nullo (`T*`);
   - `nullptr` è il valore nullo di tipo `std::nullptr_t`, convertibile in qualsiasi tipo di puntatore.
2. **Puntatori a `void`**: un puntatore di tipo `T*` può essere convertito in `void*`, ma non viceversa.
3. **Up-cast**
   - da `D*` a `B*` (dove `D` è una classe derivata di `B`);
   - analogo per i riferimento `D&` → `B&`.

4 - Conversioni definite dall'utente

Comprendono due sottotipi:

4a. Uso implicito di costruttori:

Se una classe ha un costruttore non marcato come `explicit`, può essere usato implicitamente per convertire un tipo di dato in un altro.
Es.

```
struct Razionale {
        Razionale(int num, int den = 1); //conversione implicita da int a razionale
}
```

4b. Operatori di conversione:

Permettono di definire come un tipo utente può essere convertito in un altro tipo.
Es.

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

Promozioni

Type	Promoted To	Example
`short`	`int`	`short s = 5; int i = s;`
`char`	`int`	`char c = 'A'; int i = c;`
`unsigned char`	`int`	`unsigned char u = 255; int i = u;`
`float`	`double`	`float f = 3.14f; double d = f;`
`enum`	`int`	`enum Color { Red }; int i = Red;`

Conversioni

From Type	To Type	Method	Example
`int`	`float`	Implicit	`float f = 42;`
`char`	`int`	Implicit	`int i = 'A';`
`int`	`bool`	Implicit	`bool b = 42;`
`bool`	`int`	Implicit	`int i = true;`
Derived class ptr	Base class ptr	Implicit	`Base* b = derivedPtr;`

16 - Gestione delle risorse

La gestione delle risorse garantisce che le risorse limitate siano utilizzate correttamente e rilasciate al termine del loro utilizzo.

Definizione di "risorse"

Con "risorse" indichiamo genericamente entità che sono disponibili in quantità limitata, se tali risorse vengono esaurite o mal gestite, possono compromettere il funzionamento del software.
Esempi:

- **memoria dinamica (heap)**: acquisita con `new` e rilasciata con `delete`;

- **file**: descrittori aperti tramite operazioni come `fopen` o `std::ifstream` e chiusi tramite `fclose` o la terminazione dello stream;
- **lock**: utilizzati per sincronizzare l'accesso a risorse condivise da più thread o processi;
- **connessioni di rete**: ad esempio connessioni verso server o database (DBMS);
- altri esempi includono **socket**, **handler di dispositivi** e perfino **risorse di sistema** come semafori o thread.

Tre fasi della gestione

L'interazione con una risorsa segue uno schema predefinito composto da tre fasi ordinate temporalmente:

1. **acquisizione**: la risorsa viene richiesta (es. allocazione di memoria o apertura di un file);
2. uso: la risorsa viene utilizzata per svolgere un compito specifico (es. scrittura su file o accesso a memoria);
3. rilascio: la risorsa viene restituita o liberata (es. chiusura del file o deallocazione della memoria).

Regole fondamentali

- acquisizione prima dell'uso: non si può utilizzare una risorsa prima di averla acquisita;
- rilascio dopo l'uso: una risorsa deve essere rilasciata al termine del suo utilizzo;
- divieto di uso dopo il rilascio: una risorsa non deve mai essere usata dopo essere stata rilasciata (es. accesso a puntatore "dangling");
- divieto di rilascio multiplo: non si deve rilasciare una risorsa più volte (es. doppia chiamata `delete`).

Esempi cattiva gestione delle risorse

Memory leak

Si verifica quando una risorsa (es. memoria dinamica) non viene rilasciata.

```
void memory_leak_ex() {
        int* ptr = new int(42);
        //nessuna delete -> memory leak
}
```

Dangling pointer

Si verifica quando si accede a memoria già rilasciata.

```
void dangling_pointer_ex() {
        int* ptr = new int(42);
        delete ptr;
        *ptr = 10; //errore: accesso a memoria rilasciata
}
```

Double free

Si verifica quando si tenta di rilasciare una risorsa già rilasciata.

```
void double_free_ex() {
        int* ptr = new int(42);
        delete ptr;
        delete ptr; //errore: rilascio multiplo
}
```

17 - Exception safety

Un codice si dice *exception safe* quando si comporta in maniera appropriata anche in caso di eccezioni. Questo significa che, qualora si verifichino errori gestiti tramite eccezioni, il programma:

1. non perde risorse;
2. mantiene uno stato consistente (o, nel peggiore dei casi, lascia lo stato degli oggetti manipolati distruggibile senza causare comportamenti non definiti);
3. propaga correttamente l'eccezione, permettendo al chiamante di gestirla.
   Es.

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

questo codice non è exception safe perché se la funzione `do_the_job` lancia un'eccezione, l'istruzione `delete pi` non verrà mai eseguita causando un memory leak.

Tre livelli di exception safety

1 - Livello base

Una funzione è exception safe a livello base se:

- non ci sono *perdita di risorse* (esempio: la memoria allocata dinamicamente viene sempre rilasciata anche in caso di eccezioni);
- lo stato del programma rimane *consistente* e l'oggetto su cui si lavora può essere distrutto senza causare problemi;
- le eccezioni generate vengono propagate al chiamante.

```
void example() {
    std::unique_ptr<int> ptr(new int(42)); // Risorsa gestita
    // Operazione che potrebbe lanciare un'eccezione
    risky_operation(*ptr);
    // Non serve esplicitare il rilascio: la memoria sarà gestita automaticamente
}
```

anche se `risky_operation` lancia un'eccezione, la memoria allocata tramite `std::unique_ptr` viene rilasciata automaticamente grazie alla gestione RAII (Resource Acquisition Is Inizialization) e lo stato del programma rimane consistente.

2 - Livello forte (strong exception safety)

Una funzione garantisce exception safet forte se assicura la proprietà di atomicità: o l'operazione ha successo e lo stato cambia, oppure in caso di eccezione lo stato rimane invariato.

```
void safe_insert(std::vector<int>& vec, int value) {
    std::vector<int> copy(vec); // Crea una copia del contenitore originale
    copy.push_back(value);      // Modifica la copia
    std::sort(copy.begin(), copy.end()); // Ordina la copia
    vec = std::move(copy);      // Sostituisce il contenuto originale solo se tutto
va a buon fine
}
```

Se qualcosa va storto il contenitore originale rimane intatto; lo stato di `vec` cambia solo se tutte le operazioni intermedie sono state completate con successo.
Questa garanzia è più costosa da implementare rispetto al livello base, ma è spesso desiderabile per operazioni critiche.

3 - Livello nothrow

Il livello massimo di sicurezza si raggiunge quando una funzione è dichiarata **nothrow** (ovvero garantisce che non lancerà mai eccezioni). Questo livello è fondamentale per alcune operazioni, come i distruttori e le funzioni che rilasciano risorse.

```
class SafeResource {
public:
    ~SafeResource() noexcept {
        // Garanzia che il distruttore non lanci eccezioni
        release_resource();
    }
};
```

Quando si utilizza `nothrow`:

- operazioni che non possono fallire per definizione (es. assegnamenti tra tipi built-in come `int`);
- funzioni che gestiscono completamente le eccezioni al loro interno, senza propagare errori all'esterno;
- funzioni di rilascio di risorse o distruttori: non ha senso rischiare di fallire mentre si tenta di recuperare uno stato consistente.
  N.B. le funzioni dichiarate `noexcept` non possono propagare eccezioni. Se una funzione `noexcept` lancia un'eccezione, il programma termina con un errore irreversibile.

Comportamento della libreria standard

La libreria standard garantisce che i suoi contenitori siano exception safe, ma le garanzie variano a seconda delle operazioni eseguite:

- **garanzia forte**: alcune operazioni forniscono la proprietà di atomicità, ad esempio se la `push_back()` fallisce, il contenuto del vettore rimane invariato;
- **garanzia base**: operazioni più complesse (come `assign` o `resize`) possono modificare lo stato del contenitore in caso di eccezione, ma lo stato rimane valido e consistente.

- **garanzia nothrow**: operazioni come la distruzione degli elementi o il rilascio della memoria non generano mai eccezioni.

Tecniche per raggiungere exception safety:

Vi sono tre approcci che possono essere combinati tra loro:

- evitare le eccezioni;
- uso dei blocchi `try` / `catch`;
- uso dell'idioma RAII-RRID.

Esempi:

user.cc

Codice utente che vorrebbe lavorare su alcune risorse garantendo la corretta interazione con le risorse (acquisizione, uso e rilascio) anche in presenza di errori. Intuitivamente, si vorrebbe eseguire questa sequenza di operazioni:
acquisisci risorsa r1
usa risorsa r1

acquisisci risorsa r2
usa risorse r1 e r2
restituisci risorsa r2

acquisisci risorsa r3
usa risorse r1 e r3
restituisci risorsa r3
restituisci risorsa r1

```
/* Una codifica che NON è corretta in presenza di errori */

#include "risorsa_no_exc.hh"

void codice_utente() {
  Risorsa* r1 = acquisisci_risorsa();
  usa_risorsa(r1);
  Risorsa* r2 = acquisisci_risorsa();
  usa_risorse(r1, r2);
  restituisci_risorsa(r2);
  Risorsa* r3 = acquisisci_risorsa();
  usa_risorse(r1, r3);
  restituisci_risorsa(r3);
  restituisci_risorsa(r1);
}
```

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) { //controllo se sono riuscito ad acquisire la risorsa
    // 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;
}
```

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());
  }
  Gestore_Risorsa r3;
  usa_risorse_exc(r1.get(), r3.get());
}
```

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.cpp

```
#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 "..." 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 (seconda parte del punto 6)
  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".
*/
```

18 - Smart pointers

La gestione manuale della memoria con i puntatori raw (detti anche "naked") può introdurre errori come:

- *memory leak*
- *dangling pointer*
- *double delete*
  Per risolvere questi problemi, si usa l'idioma **RAII (Resource Acquisition Is Inizialization)**.
  Questo approccio prevede che una risorsa venga gestita da un oggetto il cui costruttore la acquisisce e il cui distruttore la rilascia automaticamente.
  C++ offre tre classi principali di puntatori smart nella libreria `<memory>`:

1. `std::unique_ptr`: un puntatore che ha unicità di proprietà (owning).
2. `std::shared_ptr`: un puntatore che permette la condivisione della proprietà.

3. `std::weak_ptr`: un puntatore che non partecipa alla gestione attiva della risorsa.

1 - `std::unique_ptr`:

- gestisce una risorsa in modo esclusivo;
- non è copiabile, ma è spostabile (movable);
- rilascia automaticamente la risorsa quando esce dallo scope.

```
#include <memory>

void foo() {
    std::unique_ptr<int> pi(new int(42)); // Gestisce un intero
    *pi = 100;                            // Dereferenziazione come un normale
puntatore
} // Alla fine di foo(), pi rilascia la memoria.
```

Caratteristiche principali:

- **non copiabile**: evita duplicazioni che potrebbero portare a doppio rilascio della risorsa;
- **spostamento**: con `std::move`, la proprietà della risorsa può essere trasferita:
  `cpp void foo(std::unique_ptr<int> p); std::unique_ptr<int> ptr(new int(42)); foo(std::move(ptr)); // Trasferisce la proprietà della risorsa`
  Metodi importanti:
- `reset(raw_ptr)`: cambia la risorsa gestita, rilasciando eventualmente quella precedente;
- `get()`: restituisce il puntatore raw (la gestione resta allo `unique_ptr`);
- `release()`: rimuove le proprietà della risorsa e restituisce il puntatore raw (ora il programmatore deve gestire la memoria manualmente).
  Vantaggi:
- leggero ed efficiente;
- adatto quando è garantita l'unicità della proprietà.

2 - `std::shared_ptr`:

- gestisce una risorsa in modo condiviso tra più puntatori;
- usa un reference counter per tracciare quante copie dello shared pointer esistono;
- rilascia la risorsa solo quando il reference counter scende a 0.

```
#include <memory>

void foo() {
    std::shared_ptr<int> sp1(new int(42)); // Ref counter = 1
    {
        std::shared_ptr<int> sp2 = sp1;    // Ref counter = 2
        *sp2 = 50;                         // Modifica condivisa
    } // Ref counter = 1 (sp2 esce dallo scope)
} // Ref counter = 0, la risorsa viene rilasciata.
```

Caratteristiche principali:
copiabile e spostabile: la copia aumenta il reference counter, lo spostamento no.

```
void foo(std::shared_ptr<int> sp);
std::shared_ptr<int> sp(new int(42));
foo(sp);              // Copia condivisa, ref counter aumenta
foo(std::move(sp));   // Spostamento, ref counter non cambia
```

Metodi importanti:

- `reset()`: rilascia la risorsa e può assegnarne una nuova;
- `get()`: restituisce il puntatore raw (senza trasferire la proprietà).
- `std::make_shared`: consente di creare un `shared_ptr` in modo efficiente, allocando risorsa e blocco di controllo in un'unica operazione.

```
auto sp = std::make_shared<int>(42); // Più efficiente di `new`
```

Vantaggi:

- ideale per scenari in cui più entità devono accedere alla stessa risorsa;
- la risorsa viene rilasciata automaticamente.

3 - `std::weak_ptr`:

- è un puntatore smart che punta a una risorsa gestita da uno `shared_ptr`, senza incrementare il reference counter;
- utile per evitare cicli di riferimento (es. in strutture dati come grafi o alberi).

```
#include <memory>
#include <iostream>

void maybe_print(std::weak_ptr<int> wp) {
    if (auto sp = wp.lock()) { // Converte in shared_ptr se la risorsa è disponibile
        std::cout << *sp;
    } else {
        std::cout << "Risorsa non più disponibile";
    }
}

void foo() {
    std::weak_ptr<int> wp;
    {
        auto sp = std::make_shared<int>(42);
        wp = sp; // wp osserva sp
        maybe_print(wp); // Stampa: 42
    } // sp viene distrutto, la risorsa non è più disponibile
    maybe_print(wp); // Stampa: Risorsa non più disponibile
}
```

Vantaggi:

- risolve problemi di cicli di riferimento;
- evita memory leak in strutture con dipendenze reciproche.

Funzioni `std::make_shared` e `std::make_unique`

Le funzioni `std::make_shared` e `std::make_unique` consentono di creare puntatori smart senza usare esplicitamente `new`. Questo approccio:

- è più efficiente;
- previene problemi di exception safety.

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

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

nella prima chiamata di `bar` l'implementazione potrebbe decidere di valutare:

- prima le due `new int(42)` che sono argomenti dei costruttori dei due `shared_ptr`;
- solo dopo invocare i costruttori dei due `shared_ptr`.
  se la prima allocazione va a buon fine ma la seconda invece fallisce con eccezione, si ottiene un memory leak per la prima risorsa allocata.
  Il problema non si presenta nella seconda chiamata a `bar`, perché le allocazioni sono effettuate (implicitamente) dalla `make_shared`.

Casi particolari e linee guida

1. **cicli di riferimento**:
   - usare `std::weak_ptr` quando si creano dipendenze circolari tra shared pointers.
2. **evita `new` e `delete` diretti**:
   - le linee guida moderne del C++ sconsigliano l'uso di `new` e `delete`, privilegiando gli smart pointer.
3. **performance**:
   - usare `std::unique_ptr` dove possibile, perché è più leggero rispetto a `std::shared_ptr`.

Appello_20120202 es. 5
La classe seguente contiene errori inerenti la corretta gestione delle risorse. Individuare almeno due problemi logicamente distinti, indicando la sequenza di operazioni che porta alla loro occorrenza. Fornire quindi una soluzione alternativa e discutere brevemente i motivi per i quali tale soluzione si può ritenere corretta.

```
#include <string>
class A {
        int* pi; //puntatore ad intero
        std::string str;
        double* pd; //puntatore a double
public:
        A(const std::string& s) : pi(new int), str(s), pd(new double) { }
```

```
//costruttore
        ~A() { delete pi; delete pd; } //distruttore
};
```

risoluzione:
due problemi logicamente distinti:

1. non posso considerare il distruttore e il costruttore senza tenere presente anche il costruttore di copia e l'operatore di assegnamento (senza, gli oggetti vengono copiati con una shallow copy (vengono copiati gli *indirizzi* di pi e pd) e troviamo due oggetti A che puntano allo stesso intero e allo stesso double, se uno dei due finisce il ciclo di vita rilasciando le risorse lascerà l'altro con dei dangling pointers, se prova anche a distruggerle farà una double free);
2. se nel costruttore viene lanciata un'eccezione e l'oggetto non è ancora stato creato "completamente" la memoria allocata fino a quel momento verrà persa (memory leak) in quanto non è possibile chiamare il distruttore.

```
//modalità #1: smart pointers
#include <string>
class A {
        std::unique_ptr<int> pi; //non è possibile fare la copia
        std::string str;
        std::unique_ptr<double> pd;
public:
        A(const std::string& s) : pi(new int), str(s), pd(new double) { }

        A(const A&) = delete;
        A& operator=(const A&) = delete; //disabilito la copia

        A(A&&) = default;
        A& operator=(A&&) = default; //spostamento

        ~A() = default;
};

//modalità #2: try-catch
#include <string>
class A {
        int* pi;
        std::string str;
        double* pd;
public:
        A(const std::string& s)
                : pi(nullptr), str(s), pd(nullptr) {
                pi = new int;
                try {
                        pd = new double;
                } catch (...) {
                        delete pi;
                        throw;
                }
        }
        ~A() { //non può lanciare eccezioni
                delete pd;
```

```
                delete pi;
        }
};
```

Appello_20220907 es. 3

Il seguente codice non ha un comportamento corretto in presenza di eccezioni. Individuare almeno un problema, indicando la sequenza di operazioni che porta alla sua occorrenza. Fornire quindi una soluzione basata sull'utilizzo dei blocchi try/catch.

```
void load_and_process(const std::string& conn_params,
                                        const std::string& query) {
        Connection conn;
        Results res;

        conn.open(conn_params); //acquisizione connessione

        res.init(); //acquisizone buffer per risultati
        conn.execute(query, res); //caricamento dati
        process(res); //elaborazione dati
        res.finish(); //rilascio buffer -- non lancia eccezioni (distruzione esplicita, non è applicato l'idioma RAII)
        conn.close(); //rilascio connessiome -- non lancia eccezioni    (distruzione esplicita, non è applicato l'idioma RAII)
}
```

risoluzione:

```
void load_and_process(const std::string& conn_params,
                                        const std::string& query) {
        Connection conn;
        Results res;

        conn.open(conn_params); // se l'acquisizione dà errore (quindi non viene acquisita la risorsa) non c'è bisogno di liberarla
        try {
                res.init(); //acquisizione
                try {
                        conn.execute(query, res);
                        process(res);
                        res.finish();
                        conn.close();
                } catch (...) {
                        res.finish();
                        throw;
                }
        } catch (...) {
                conn.close();
                throw;
        }
}
```

19 - Template

Template di funzione

Un template di funzione consente di scrivere un modello parametrico per una funzione, rendendola generica e adatta a lavorare con diversi tipi di dato.

Dichiarazione e Definizione

Un template di funzione si dichiara e definisce con la parola chiave `template`, seguita da una lista di parametri di template racchiusa tra parentesi angolate `<>`. Questi parametri rappresentano tipi di dato generici.

```
//dichiarazione
template <typename T>
T max(T a, T b);

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

Parametri del template

I parametri di un template possono rappresentare:

1. **tipi generici** (ad esempio, `T` in `typename T`);
2. **valori** (ad esempio, interi o puntatori);
3. **template di altri template** (template nidificati);
   Sintassi:

- la parola chiave `typename` o `class` può essere usata per indicare un tipo generico. Le due parole chiave sono equivalenti, ma `typename` è preferibile per coerenza semantica.
- i nomi dei parametri di tipo sono convenzionalmente maiuscoli.

```
template <typename T, int N>
T arrayMax(T (&arr)[N]) {
  T maxVal = arr[0];
  for (int i = 1; i < N; ++i) {
    if (arr[i] > maxVal) {
      maxVal = arr[i];
    }
  }
  return maxVal;
}
```

Istanziazione dei template di funzione

Quando si utilizza un template, il compilatore crea automaticamente una istanza del template con i tipi specificati.

Istanziazione implicita

Il compilatore deduce automaticamente i tipi degli argomenti passati alla funzione.

```
int maxInt = max(10, 20);     // T dedotto come int
double maxDouble = max(3.5, 2.1); // T dedotto come double
```

Istanziazione esplicita

È possibile specificare esplicitamente i tipi nella chiamata alla funzione.

```
int maxInt = max<int>(10, 20);
```

Problemi di deduzione

La deduzione dei tipi fallisce se gli argomenti non corrispondono univocamente a un tipo.

```
int result = max(10.5, 20); // Errore: T non può essere dedotto
int result = max<int>(10.5, 20); // Corretto: forzo T = int
```

Specializzazione esplicita

La specializzazione esplicita consente di fornire una definizione specifica per determinati tipi.

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

// Specializzazione per const char*
template <>
const char* max<const char*>(const char* a, const char* b) {
  return strcmp(a, b) > 0 ? a : b;
}
```

N.B. la lista vuota di parametri indica una specializzazione totale per uno specifico tipo.

Istanziazione Esplicita

È possibile richiedere al compilatore di istanziare un template senza usarlo direttamente nel codice.

```
// Dichiarazione di istanziazione
extern template int max<int>(int, int);

// Definizione di istanziazione
template int max<int>(int, int);
```

N.B.

Esiste una differenza sostanziale tra un template di funzione e le sue possibili istanziazioni

- un template di funzione non è una funzione (è un "generatore" di funzioni);
- una istanza di un template di funzione è una funzione.

Template di classe

Un template di classe consente di scrivere un modello parametrico per una classe, rendendo possibile generare classi per tipi diversi.

Dichiarazione e Definizione

Un template di classe viene definito con la stessa sintassi dei template di funzione, ma con una classe al posto di una funzione.

```
template <typename T>
class Stack {
private:
  std::vector<T> elements;

public:
  void push(const T& element) {
    elements.push_back(element);
  }
  void pop() {
    elements.pop_back();
  }
};
```

Istanziazione di template di classe

Per i template di classe, i parametri non vengono dedotti automaticamente; devono essere specificati esplicitamente.

```
Stack<int> intStack;       // Istanza per int
Stack<std::string> strStack; // Istanza per std::string
```

Deduzione con `auto`:

```
auto copyStack = intStack; // Deduzione del tipo
```

Specializzazione dei template di classe

Come per i template di funzione, è possibile creare specializzazioni **totali** e **parziali**.

Specializzazione totale

Una definizione completa per un tipo specifico.

```
template <>
class Stack<bool> {
private:
  std::vector<unsigned char> bits; // Ottimizzazione per bool
```

```
public:
  void push(bool value) { /* ... */ }
  void pop() { /* ... */ }
};
```

Specializzazione parziale

Una definizione per un sottoinsieme di tipi.

```
template <typename T>
class Stack<T*> { // Specializzazione per puntatori
private:
  std::vector<T*> elements;

public:
  void push(T* element) { elements.push_back(element); }
  void pop() { elements.pop_back(); }
};
```

Altri tipi di template

Template di Alias

Un template di alias consente di creare alias per tipi complessi.

```
template <typename T>
using Vec = std::vector<T, std::allocator<T>>;
```

Template di variabile

Un template di variabile consente di creare costanti parametrizzate.

```
template <typename T>
constexpr T pi = T(3.1415926535897932385);
```

21 - Programmazione generica

I template in C++ sono il fondamento della programmazione generica. Permettono di scrivere codice riutilizzabile e generico che funzionano con tipi diversi. Il termine "polimorfismo statico" deriva dal fatto che il tipo specifico viene risolto a tempo di compilazione, a differenza del polimorfismo dinamico che avviene a runtime.
La programmazione generica è una metodologia di programmazione fortemente basata sul polimorfismo statico: i maggiori benefici si ricavano da un certo numero di template progettati in maniera coordinata per fornire interfacce comuni ed estendibili.

Contenitori

- **contenitore** := classe che ha lo scopo di contenere una collezione di oggetti; tipicamente realizzati mediante template di classe che si differenziano a seconda dell'organizzazione degli elementi e

delle operazioni fondamentali che si vuole supportare su di essi.

Contenitori sequenziali

Forniscono l'accesso ad una sequenza di elementi organizzati in base alla loro posizione; il loro ordinamento non è stabilito secondo alcun criterio, ma viene dato dalle operazioni di inserimento e rimozione. I contenitori sequenziali standard sono:

- `std::vector<T>`
  - sequenza di `T` di dimensione variabile (a tempo di esecuzione) memorizzati in modo contiguo;
  - accesso a qualunque elemento in tempo costante;
  - inserimenti e rimozioni sono efficienti se fatti in fondo alla sequenza.
- `std::deque<T>`
  - "double-ended queue" è una coda a doppia entrata;
  - inserimenti e rimozioni possono essere effettuati sia in fondo alla sequenza che all'inizio (possibile in quanto non abbiamo la certezza che gli elementi siano memorizzati in modo contiguo)
  - accesso a qualunque elemento in tempo costante.
- `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;
  - doppia concatenazione = scorrere la lista avanti e indietro (bidirezionale);
  - per accedere a un elemento bisogna "raggiungerlo" seguendo il link della lista;
  - inserimenti e rimozioni effettuati in tempo costante nella posizione corrente in quanto non serve spostare elementi.

  Oltre a questi, esistono gli "pseudo-contenitori":
- `std::array<T, N>`
  - sequenza di T di dimensione N, fissata a tempo di compilazione (con N parametro valore non typename);
  - corrisponde ad un array del linguaggio ma senza le problematiche relative al type decay e permette di conoscere facilmente il numero di elementi;
- `std::string`
  - sequenza di caratteri (char);
  - è un alias per l'istanza `std::basic_string<char>`, può essere istanziato anche con altri tipi carattere, per cui abbiamo altri alias come:
    - `std::wstring` → `wchar_t`
    - `std::u16string` → `char16`
    - `std::u32string` → `char32`
- `std::bitset<N>`
  sequenza di esattamente N bit (con N parametro valore).

Operazioni

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);
- altri operatori specifici.

Algoritmi generici: dai tipi ai concetti

I contenitori e gli pseudo-contenitori visti sono caratterizzati da interfacce simili, ma non esattamente identiche, inoltre non hanno tutti i servizi che si aspetta di poter utilizzare (es. riordinare gli elementi di un vector); infatti la libreria standard li implementa come *algoritmi generici*, non sono pensati per lavorare con tipi di dato specifici, ma per lavorare su concetti astratti ed essere applicabili a tutti i tipi di dato.

Esempio: si vuole implementare un algoritmo che cerca un elemento, con un certo valore, all'interno di un contenitore. Visto in "astratto" può essere applicato ad una qualunque sequenza i cui elementi possano essere scorsi, dall'inizio alla fine, e confrontati con l'elemento cercato (:= il tipo contenitore può essere sostituito dal "concetto" astratto di sequenza).
Una sequenza può essere rappresentata da una *coppia di iteratori* → **first** e **last**.
[first, last)
(first compreso, last escluso)
iteratori := concetto "astratto"
Esempio di iteratore ⇒ tipo puntatore ad un elemento contenuto in un array:

```
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";
}
```

Questo algoritmo funziona solo per puntatori a interi, per aumentare l'applicabilità lo rendiamo templatico:

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

```
  return last;
}
```

Scegliere `T*` (puntatore ad un tipo qualunque) è una scelta *limitante*; possono esserci altri tipi di dato, oltre ai puntatori:

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

⇒ sostituiamo `T*` con un altro parametro di template `Iter`, che deve fornire il concetto di iteratore (non per forza puntatore).
Requisiti per istanziare correttamente `Iter` in questo algoritmo:

- `Iter` deve supportare la copia (passato e restituito per valore);
- `Iter` deve supportare il confronto binario (`first != last`);
- `Iter` deve supportare il preincremento (`++first`) per avanzare di una posizione nella sequenza;
- `Iter` deve consentire la dereferenziazione (`*first`), per poter leggere il valore puntato;
- il tipo dei valori puntati da `Iter` deve essere confrontabile con il tipo T (usando l' `operator==`).
  Può essere utilizzato qualsiasi tipo di dato concreto (puntatore o meno) a patto che soddisfi questi requisiti.

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 (:= le operazioni disponibili su di esso e la sua semantica).
Gli algoritmi generici hanno applicabilità generale; in particolare, non ci sono algoritmi specifici per ogni contenitore, ma ogni contenitore fornisce (attraverso i suoi iteratori) la possibilità di essere visto come sequenza.

Contenitori Associativi

Contenitori che organizzano gli elementi al proprio interno in modo da facilitarne la ricerca (non in base alla posizione) in base al valore di una chiave.
I contenitori associativi della libreria standard si differenziano in base a tre caratteristiche binarie:

- `set<Key, Cmp>` //ordered è implicito dove non è scritto
- `multiset<Key, Cmp>`
- `map<Key, Mapped, Cmp>`
- `multimap<Key, Mapped, Cmp>`
- `unordered_set<Key, Hash, Equal>`
- `unordered_multiset<Key, Hash, Equal>`
- `unordered_map<Key, Mapped, Hash, Equal>`
- `unordered_multimap<Key, Mapped, Hash, Equal>`

Caratteristiche Binarie

1. Presenza di informazioni supplementari, oltre la chiave. (es. key + altro: `map`, solo key: insieme);
2. possibilità di memorizzare più elementi con lo stesso valore per la chiave (es. versioni `multi`);
3. struttura dati usata per il contenitore (modalità di implementazione interna, sfruttano un criterio di ordinamento sulle chiavi), esempio:
   - le versioni `sorted` utilizzano alberi di ricerca bilanciati, basati su un criterio di ordinamento(`Cmp`) definito sul tipo `Key`;
   - le versioni `unsorted` utilizzano le tabelle hash, basate su una funzione di hashing `Hash` e una relazione equivalente `Equal` per risolvere i conflitti di chiave.

```
#include <iostream>
#include <map>
#include <iterator>
#include <string>

//es: programma che legge dallo standard input delle stringhe e conta quante volte quella stringa compare
int main() {
        std::map<std::string, unsigned long> freq_map; //chiave stringa, valore quante volte compare la stringa
        std::istream_iterator<std::string> i_first(std::cin);
        std::istream_iterator<std::string> i_last;

        for ( ; i_first != i_last; ++i_first) {
                const auto& s = *i_first; //stringa s ottenuta deferenziado i_first
                //voglio controllare se s è nella mappa, se non c'è la inserisco la inserisco incrementando il contatore a uno, se già è presente incremento il contatore senza inserire di nuovo
                //se non forniamo noi il criterio di ordinamento quello di default è l'operator< sulla chiave (in questo caso in base alla string -> ordine alfabetico)
                //METODO NORMALE: trova se esiste nel contenitore un elem con questa chiave: .find(str);
                auto it = freq_map.find(s); //restituisce un iteratore che punta all'elemento, se punta all'ultimo elemento del contenitore, vuol dire che l'elem non è stato trovato
                if (it == freq_map.end()) {
                        //inserisci e incrementa
                        //freq_map.insert(std::pair<std::string, unsigned long>(s,1))
                        freq_map.insert(std::make_pair(s, 1));
                } else {
                        //incrementa
                        ++(it->second); //++(*it).second
                }
        }
        //METODO SMART: possiamo usare [] come per gli array, ma invece di un indice numerico ci mettiamo il valore della chiave
        for ( ; i_first != i_last; ++i_first) {
                const auto& s = *i_first;
                if (it == freq_map.end()) {
                        //freq_map.insert(std::make_pair(s, 1));
                        ++freq_map[s]; //come valore mette il valore che si ottiene intuitivamente con il costruttore di default del tipo mappato: unsigned long() value
```

```
initialization -> 0, ma noi vogliamo 1, quindi incrementiamo
                } else {
                        ++freq_map[s]; //l'operator[] quando non trova l'elem lo
inserisce automaticamente
                }
        }
        //VERO METODO SMART
        for ( ; i_first != i_last; i_first++){
                const auto& s = *i_first;
                ++freq_map[s];
        }

        for (const auto& p : freq_map) {
                std::cout << "La parola " << p.first
                << " occorre numero " << p.second << " volte\n";
        }
}
```

Lezione 20-11-2024

```
#include <algorithm>
#include <iostream>
template <typename Iter, typename UnaryPred>
Iter find_if(Iter first, Iter last, UnaryPred pred) { //non vogliamo passare un
vector, su qualunque cosa è possibile definire una coppia di iteratori lo facciamo
        for ( ; first != last; ++first) { //iteratori modellati su sintassi come
puntatori
                if (pred(*first)) //dereferenzio l'iteratore per arrivare
all'elemento
                        return first;
        }
        return last; //vuol dire che l'elemento non è stato trovato
}

template <typename Iter, typename UnaryPred>
bool all_of(Iter first, Iter last, UnaryPred pred) { //il predicato deve essere vero
per tutti gli elementi
        for ( ; first != last; ++first) {
                if (not pred(*first))
                        return false;
        }
        return true;
}

template <typename Iter, typename UnaryPred>
bool any_of(Iter first, Iter last, UnaryPred pred) { //almeno un elemento uguale al
pred
        for ( ; first != last; ++first) {
                if (not pred(*first))
                        return true;
        }
        return false;
}
```

```cpp
template <typename Iter, typename UnaryPred>
bool none_of(Iter first, Iter last, UnaryPred pred) { //nessun elemento uguale
}

////////////////////////////////////////////////////////////////////

//cerca nel range [first,last) se un qualunque elemento appare nella seconda sequenza [s.first, s.last)
template <typename Iter>
Iter find_first_of(Iter first, Iter last,
                                    Iter first2, Iterlast2) { //do per scontato che gli iteratori siano dello stesso tipo -> sbagliato
}

template <typename Iter, typename Iter2>
Iter find_first_of(Iter first, Iter last,
                                    Iter2 first2, Iter2 last2) {
        for ( ; first != last; first++) {
                for (Iter2 it2 = first2 ; it2 != last2; it2++){ //la seq esterna la scorro una volta sola da inizio a fine, la seq interna, ogni volta che faccio un ciclo, devo scorrere dall'inizio alla fine, percio creiamo un nuovo iteratore cosi da non cambiare first2, perdendo l'indice del dato che vogliamo confrontare senza poter tornare indietro
                if (*first == *it2)
                        return first;
                }
        }
        return last;
}

template <typename Iter, typename Iter2, typename BinaryPred>
Iter find_first_of(Iter first, Iter last,
                                    Iter2 first2, Iter2 last2,
                                    BinaryPred pred) {
        for ( ; first != last; first++) {
                for (Iter2 it2 = first2 ; it2 != last2; it2++){
                if (pred(*first, *it2))
                        return first;
                }
        }
        return last;
}

//esempio di predicato binario
bool doppio_di(int i, int j) { //predicato binario
        return (i == 2*j); //restituisce qualcosa che si possa convertire in booleano
}

template <typename T>
struct Doppio_di {
        bool operator() (const T& i, const T& j) const { //non so di che tipo sono percio li passo per riferimento perche potrebbero essere troppo pesanti da copiare
                return i == 2*j;
};
```

```
////////////////////////////////////////////////////////////////////////

bool maggiore_di_10000 (int i) {
        return i >10000;
}

//oggetti di funzione
struct Maggiore_di_N {
        int N_;

        Maggiore_di_N(int N) : N_(N) {}

        bool operator() (int i) {
                return i > 10000;
        }
};

void pippo(const std::vector<int>& i) {
        if (std::all_of(vi.begin(), vi.end(), maggiore_di_10000))
                std::cout << "caz";
        if (std::all_of(vi.begin(), vi.end(), Maggiore_di_N(10000)))
                std::cout << "caz";
}

////////////////////////////////////////////////////////////////////////

template <class InputIt, class OutpuIt> //class == typename, class non è sempre
appropriato perche potrebbe non essere una classe
OutputIt copy(InputIt first, InputIt last, //mi definisce la seq
                        OutputIt d_first) { //d_first è la destinazione e deve essere
abbastanza grande
        //leggere dalla prima seq
        for ( ; first != last; ++first) {
                //scrivere nella seconda sequenza -> prendi first, dereferenzialo,
usa              l'assegnamento
                *d_first = *first; //destra leggo, sinistra scrivo
                //andare al prossimo elemento
                ++d_first;
        }
        return d_first;//torna per valore e so qual è la posizione giusta dalla quae
ripartire per scrivere
}

//copy_if non copia tutto, ma solo ciò che rispetta il predicato
template <class InputIt, class OutpuIt, class UnaryPred>
OutputIt copy_if (InputIt first, InputIt last,
                        OutputIt d_first, UnaryPred pred) {
        for ( ; first != last; ++first) {
                if (pred(*first)) {
                        *d_first = *first;
                        ++d_first;
                }
        }
        return d_first;
}
```

```
//se non vogliamo sovrasrivere dobbiamo utilizzare un atro tipo di iteratori -> inseritori, iteratori di output diversi che non sovrasrivono ma vanno ad inserire

//back_insert utilizzano un contenitore, non una coppia di iteratori e va a creare l'iteratore per andare ad inserire gli elementi nella posizione finale
//front_insert inseriscono all'inizio, utilizzato per code, liste, ecc..
//insert_iterator inseriscono in una data posizione

void pippo(const std::vector<int>& i) {
        if (std::all_of(vi.begin(), vi.end(), maggiore_di_10000))
                std::cout << "caz";
        if (std::all_of(vi.begin(), vi.end(), Maggiore_di_N(10000)))
                std::cout << "caz";

        std::vector<double> vd;

        std::copy(vi.begin(), vi.end(), vd.end())

        std::copy_if(vi.begin(), vi.end(), vd.end(), Maggiore_di_N(50)); //undefined behaviour, vd.end() è uno dopo la fine, proviamo a scrivere in una cella di memoria che non ci appartiene; brute force solution: creo vector con una determinata dimensione:
        std::vector<double> vd2(1000, 0.0);
        std::copy_if(vi.begin(), vi.end(), vd.begin(), Maggiore_di_N(50));
        //spreco di risorse

        std::vector<double> vd3(vi.size(), 0.0);
        std::copy(vi.begin(), vi.end(), vd3.begin());

        std::back_insert_iterator<std::vector<double>> out(vd3); //si scrive troppa roba perche le classi templatiche non sopportano la deduzione automatica dei parametri dei template -> cosa che invece fanno le funzioni templatiche:
        auto out = std::back_inserter(vd); //+ comodo usare le funzioni di supporto
}
```

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>

int main() {
//legge da standard input stringhe e fa l'echo per lo standard output, ma mette una stringa per ogni riga
        std::istream_iterator<std::string> first(std::cin);
        std::istream_iterator<std::string> last; //end è data di default
        std::ostream_iterator<std::string> out(std::cout, "\n");//voglio vederlo come sequenza di stringhe, \n è il separatore
        std::copy(first, last, out);
        std::copy_if(first, last, out, lunga)
}

////////////////////////////////////////////////////////////////////////////7
```

```
//stringhe piu lunghe di 10
bool lunga(const std::string& s) {
        return s.size() >= 10;
}
```

23 - Iteratori

Codice 22-11-2024

```
#include <algorithm>
#include <iostream>

#include <iterator>
#include <string>

////output stream iterator definiti neell'header file iterator, suddiviso in porzioni separate incluse a loro volta

bool lunga (const std::string& s){
        return s.size() >= 10;
}

template< typename InputIt, typename OutputIt, typename UnaryOp>
OutputIt trasform (InputIt first1, InputIt last1,
                                   OutputIt d_first, UnaryOp unary_op);


template< class InputIt1, class InputIt2, class OutputIt, class BinaryOp>
OutputIt trasform (InputIt first1, InputIt last1, InputIt first2,
                                   OutputIt d_first, BinaryOp binary_op) {//due seq tale che la prima non può essere piu corta della seconda, in modo da lavorare solo sulla sequenza in comune ed avere piu efficienza
        for ( ; first1 != last1; first1++) { //la seconda seq è lunga almeno quanto la prima
                *d_first = binary_op(*first1, *first2);
                ++first2; //x passare al successivo
        }
        return d_first; //ritorno l'operatore di output (quasi tutti gli algo che usano l'operatore di output lo ritornano -> per poter concatenare le operazioni)
}

int main() {
        std::istream_iterator<std::string> first(std::cin);
        std::istream_iterator<std::string> last;

        std::ostream_iterator<std::string> out(std::cout, "\n");

        std::copy_if(first, last, out, lunga);
}
```

Iteratori

Il concetto di iteratore fornisce modi per rappresentare le sequenze, indipendentemente dal tipo utilizzato per l'implementazione.
Esistono 5 categorie che si differenziano per le operazioni supportate:

- **iteratori di input**:
  operazioni consentite

  - `++iter`, avanza di una posizione
  - `iter++`, preferire il prefisso
  - `*iter`, accesso (in read) 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 (verifica se siamo alla fine della sequenza)
  - `iter1 != iter2` confronto per disuguaglianza
    Esempio: iteratori definiti sugli stream di input `std::istream`, attraverso i quali è possibile leggere i valori sullo stream.

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

int main() {

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

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

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

  in questo caso, 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 con costruttore di default.
  Gli iteratori di input non sono riavvolgibili (one shot), ovvero non consentono di scorrere più volte la stessa sequenza, incontriamo un comportamento indefinito. L'operazione di avanzamento consuma l'input letto e per poterlo rileggere occorre salvarlo da qualche altra parte.
- **iteratori forward**:
  effettuano tutte le operazioni degli input iterator, cambia però la semantica. L'incremento non invalida altri iteratori, sono riavvolgibili e consentono di scorrere più volte la stessa sequenza. Se il tipo dell'elemento indirizzato è modificabile, è possibile utilizzarli anche per scrivere sulla sequenza.
  Esempio: iteratori del `std::forward_list`

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

  // Modifica gli elementi della lista
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
```

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

//Possiamo usare la copy sugli op del forward list? Si, perche sanno fare tutto ciò che sanno fare gli input iterator (duck type system)

- **iteratori bidirezionali**:
  effettuano tutte le operazioni supportate dagli iteratori forward e consentono di spostarsi all'indietro sulla sequenza con gli operatori di decremento:
  - `- -iter`, si sposta alla posizione precedente
  - `iter - -` (preferire il prefisso)
    Resi disponibili per esempio nel `std::list`, `std::set`, `std::map`...

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

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

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

  // Stampa i valori all'indietro
  // Nota: l'uso di "auto" mi risparmia dal dover scrivere il tipo
  // dell'iteratore usato, che sarebbe std::list<int>::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<int>::const_reverse_iterator
  for (auto i = lista.crbegin(); i != lista.crend(); ++i)
        std::cout << *i << std::endl;
}
```

- **iteratori random access**:
  effettuano tutte le operazioni supportate dagli iteratori bidirezionali e consentono di spostarsi liberamente nella sequenza
  operazioni:

- `iter += n`, sposta iter di n posizioni (avanti: n positivo, indietro: n negativo)
- `iter -= n`, analogo, ma direzione opposta
- `iter + n`, calcola un iteratore spostato di n posizioni senza modificare iter
- `n + iter`, equivalente
- `iter - n`, analogo, direzione opposta
- `iter[n]`, equivalente a `*(iter + n)`
- `iter1 - iter2`, calcola la distanza tra due iteratori, ovvero quanti elementi dividono le due posizioni (definiti sulla stessa sequenza)
- `iter1 < iter2`, true: se iter1 occorre prima di iter2 (sulla stessa sequenza)

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

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

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

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

- **iteratori di output**:
  permettono solamente di scrivere gli elementi di una sequenza, l'operazione di scrittura deve essere fatta una volta sola, successivamente bisogna incrementare per riposizionare l'iteratore e prepararlo per la scrittura successiva.
  operazioni:
  - `++iter`, avanza di una posizione la sequenza
  - `*iter`, accesso in sola scrittura all'elemento corrente
    Non è possibile confrontare tra loro iteratori di output (non necessario).
    Assumono di default che ci sia spazio nella sequenza per scrivere, compito di chi usa l'iteratore garantirlo (! → undefined behavior).

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

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
```

```
    }

  }
```

Gli iteratori forward, bidirezionali e random access soddisfano tutti i requisiti se gli oggetti "puntati" sono accessibili anche in scrittura.

```
template <typename Iter, typename Iter2>
Iter find_first_of(Iter first, Iter last,
                                  Iter2 first2, Iter2 last2) {
        for ( ; first != last; first++) {
                for (Iter2 it2 = first2 ; it2 != last2; it2++){
                if (*first == *it2)
                        return first;
                }
        }
        return last;
}
```

Utilizzano gli iteratori forward, non possono utilizzare input iterator in quanto ha bisogno di riavvolgere la sequenza.

```
template <typename ForwardIt, typename T>
void replace(ForwardIt first, ForwardIt last, )
```

replace lavora in lettura e scrittura su una sequenza

```
template <class InputIt1, class InputIt2, class OutputIt>
OutputIt merge(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2,
                          OutputIt d_first);
```

`std::merge` richiede input ordinati per garantire che l'output sia ordinato.

Template di classe `std::iterator_traits`

Alias di tipo per gli iteratori
Quando scriviamo algoritmi generici che sfruttano il concetto di iteratore, abbiamo bisogno di usare nomi canonici per accedere ai tipi associati agli iteratori. Tuttavia, non tutti gli iteratori sono implementati come classi e, quindi, non sempre è possibile accedere direttamente ai loro tipi membri. Per questo motivo, la libreria standard fornisce il template di classe `std::iterator_traits`, che consente di interrogare gli iteratori in modo uniforme, indipendentemente dal fatto che siano classi o tipi primitivi come puntatori.

Un iteratore implementato come classe dovrebbe fornire i seguenti alias di tipo:

- `value_type`: il tipo di dati ottenuto dereferenziando l'iteratore.
- `reference`: il tipo riferimento associato a `value_type`.
- `pointer`: il tipo puntatore associato a `value_type`.

- `difference_type`: un tipo intero con segno che rappresenta la distanza tra due iteratori.
- `iterator_category`: una categoria che indica le operazioni supportate dall'iteratore (input, forward, bidirectional, random access, ecc.).
  Oss.

1. `const_reference` e `const_pointer`, perché è l'iteratore che decide se il `value_type` è o meno in sola lettura;
2. 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; sono definiti nella libreria standard come:
   - `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` bidirection_iterator_tag `può essere convertito implicitamente (up-cast) in un` forward_iterator_tag `o ad un` input_iterator_tag `(un bidirectional è accettabile se viene richiesto un forward, ma non il contrario). Per fornire tutti gli alias di tipo non possiamo utilizzare la tecnica dei contenitori standard, perchè tra gli iteratori ci sono anche tipi che non sono classi, come i puntatori; perciò non possiamo utilizzare:` Iter::value_type `. Aggiriamo il problema usando il template di classe` std::iterator_traits `, in questo modo non interroghiamo direttamente il tipo iteratore, ma la classe traits ottenuta con l'istanziazione del template con quel tipo di iteratore (es.` std::iterator_traits::value_type`).
   Esempio traits:

```
tmp = *iter; //value_type dell'iteratore
typename std::iterator_traits<Iter>::value_type tmp = *iter;
//risolvibile come:
auto tmp = *iter;
```

`std::iterator_traits` è uno degli esempi di uso di classi "traits", ovvero tipi di dato che hanno lo scopo di fornire informazioni ("traits", caratteristiche) di altri tipi di dato. In particolare, consento di effettuare analisi di introspezione anche sui tipi built-in.
Altri esempi:

- `std::numeric_limits` per interrogare i tipi numerici per ottenere informazioni come i valori minimi e massimi rappresentabili, la signedness...
- `std::char_traits` per interrogare i tipi carattere.

Definizione generale

```
template <typename Iter>
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;
};
```

Questa definizione è utilizzata per gli iteratori personalizzati dall'utente:

- il template `iterator_traits` accede ai tipi membri definiti dall'iteratore stesso, come `iterator_category`, `value_type`...
- si assume che il tipo `Iter` (definito dall'utente) abbia già dichiarato questi tipi come membri. In pratica, questo permette a `iterator_traits` di delegare il lavoro di introspezione (cioè determinare i tipi necessari) direttamente all'implementazione dell'iteratore personalizzato.

Quando `Iter` è un puntatore (ad esempio `int*`), la definizione generica non può funzionare, perchè i puntatori non hanno membri come `iterator_category`, `value_type`, ecc. Per la gestione di questi casi si utilizzano le specializzazioni parziali:

Caso 1: Puntatore normale

```
template <typename T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                         pointer;
  typedef T&                         reference;
};
```

- `iterator_category` viene impostata a `random_access_iterator_tag`, perchè i puntatori sono considerati iteratori di tipo "random access" (accesso diretto a qualsiasi elemento con `[]`, `+`, `-`,...);
- `value_type` tipo di dati a cui il puntatore punta (`T`);
- `difference_type` è `ptrfidd_t`, ovvero la differenze tra due puntatori;
- `pointer` è il puntatore stesso (`T*`)
- `reference` è il riferimento al tipo puntato (`T&`).

Caso 2: Puntatore const

```
template <typename T>
struct iterator_traits<const T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef const T*                   pointer;
  typedef const T&                   reference;
};
```

24 - Callable

Concetto

Molti algoritmi generici disponibili dalla libreria standard sono forniti in due differenti versioni: la seconda è parametrizzata rispetto a una *policy*.
es. `std::adjacent_find` che ricerca all'interno di una sequenza la prima occorrenza di due elementi adiacenti ed equivalenti
due dichiarazioni:

```
template <typename FwdIter>
FwdIter adjacent_find(FwdIter first, FwdIter last) {
        if ( first == last )
                return last;
        FwdIter next = first;
        ++ next;
        while ( next != last ) {
                if (* first == * next ) // test di equivalenza
                        return first;
                first = next;
                ++ next;
        }
        return last;
}
```

il predicato binario utilizzato per il controllo di equivalenza degli elementi è l' `operator==`

```
template <typename FwdIter, typename BinPred>
FwdIter adjacent_find(FwdIter first, FwdIter last, BinPred pred) {
        if ( first == last )
                return last;
        FwdIter next = first;
        ++ next;
        while ( next != last ) {
                if ( pred (* first, * next )) // test di equivalenza
                        return first;
                first = next;
                ++ next;
        }
        return last;
}
```

consente di utilizzare un qualunque tipo di dato, a condizione che questo si comporti come predicato binario definito sugli elementi della sequenza.

Un **callable** è qualsiasi cosa che può essere invocata con la sintassi di una funzione:
`fun(arg1, arg2, ...);`
In particolare, include:

- **puntatori a funzione**: un puntatore a funzione è il modo più semplice e diretto per definire un callable.

  ```
  bool pari(int i) {
      return i % 2 == 0;
  }
  ```

```
int main() {
    bool (*fun_ptr)(int) = &pari;  // Puntatore alla funzione "pari"
    std::cout << fun_ptr(4);       // Invocazione attraverso il puntatore
}
```

Il nome della funzione stessa può essere usato direttamente come parametro dove è richiesto un callable:

```
std::find_if(v.begin(), v.end(), pari);
```

- **oggetti funzione**: sono oggetti che possono essere usati come callable grazie alla definizione dell'operatore `operator()` all'interno della classe.

```
struct Pari {
    bool operator()(int i) const {  // L'oggetto Pari è "callable"
        return i % 2 == 0;
    }
};

int main() {
    Pari pari;
    if (pari(4)) {  // Usato come una funzione
        std::cout << "Numero pari!";
    }
}
```

Vantaggi:

1. *ottimizzazione*: il compilatore può ottimizzare meglio le chiamate agli oggetti funzione rispetto ai puntatori a funzione. Per esempio, può espandere il codice in modo "inline".
2. *personalizzazione*: gli oggetti funzione possono avere stato (dati membro) e comportamento personalizzato, mentre una funzione normale no.
   Esempio di stato in un oggetto funzione:

```
struct MaggioreDi {
    int soglia;
    MaggioreDi(int s) : soglia(s) {}

    bool operator()(int i) const {
        return i > soglia;
    }
};

int main() {
    MaggioreDi maggiore_di_10(10);
    std::cout << maggiore_di_10(15);  // Restituisce true
}
```

- espressioni lambda (c++11):

Espressioni Lambda

Capita spesso che una funzione debba essere fornita come callable ad una singola invocazione di un algoritmo generico, in questi casi fornire la definizione presenta svantaggi come: inventare un nome appropriato e fornire la definizione in un punto diverso del codice rispetto all'unico punto di uso, perciò ricorriamo alle funzioni lambda.
Le **lambda expressions** (o funzioni lambda) sono una sintassi comoda per definire oggetti funzione *anonimi* direttamente nel punto in cui sono necessari.

```
auto lambda = [](int i) { return i % 2 == 0; };
std::cout << lambda(4);  // Stampa true
```

Struttura

`[capture](parametri) -> tipo_di_ritorno { corpo };`

- `[]` : capture list, definisce quali variabili locali possono essere "catturate" dalla lambda;
- `(parametri)` : lista dei parametri della funzione (opzionale);
- `-> tipo_di_ritorno` : specifica il tipo di ritorno (opzionale, può essere dedotto automaticamente);
- `{ corpo };` : il corpo della funzione

```
void foo(const std::vector& v) {
        auto iter = std::find_if(v.begin(), v.end(),
                                                                [](const long& i) {
                                                                        return i % 2
== 0;
                                                                });
        // ... usa iter
}
```

la lista delle catture è vuota, il tipo di ritorno è omesso in quanto dedotto dal return.
E' possibile specificarlo con il **trailing return type**:
`[]( const long & i ) -> bool { return i % 2 == 0; };`
L'uso dell'espressione lambda all'interno di `std::find_if` corrisponde all'esecuzione di queste operazioni:

1. definizione di una classe anonima per oggetti funzione
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 anonimo, avente il tipo della classe anonima, da passare alla invocazione. Ovvero:

```
struct Nome_Univoco {
        bool operator()(const long& i) const { return i % 2 == 0; }
};
auto iter = std::find_if(v.begin(), v.end(), Nome_Univoco());
```

Lista delle Catture

Può essere usata quando l'espressione deve poter accedere a variabili locali visibili nel punto in cui viene creata (diverso dal punto in cui verrà invocata):

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

è equivalente ad una classe nella quale le variabili catturate sono memorizzate in dati membro.

Catture di variabili

Quando la lambda usa variabili locali definite al di fuori di essa, queste devono essere catturate nella **capture list** (`[]`):

- *cattura per valore* (`=`): le variabili sono copiate all'interno della lambda;
- *cattura per riferimento* (`&`): la lambda accede direttamente alle variabili originali.

```
int soglia = 10;
auto lambda = [soglia](int i) { return i > soglia; }; //cattura per valore
auto lambda_ref = [&soglia](int i) { return i > soglia; }; //cattura per riferimento
```

Catture implicite

- `[=]` : cattura per valore ogni variabile locale usata nel corpo
- `[&]` : cattura per riferimento **ogni variabile locale usata nel corpo**
- `[=, &pippo]` : cattura per valore, tranne pippo catturato per riferimento
  preferire le catture esplicite.

```
int soglia = 10;
int moltiplicatore = 2;
auto lambda = [=](int i) { return i > soglia * moltiplicatore; };
```

Funzioni Lambda con nome

E' possibile dare un nome alle funzioni lambda con `auto`. Esempio: diamo un nome all'oggetto lambda per poterlo utilizzare più volte.

```
void copia_corte ( const std::vector<std::string>& v,
                                const std::list<std::string>& l,
                                unsigned max_size ) {
        auto corta = [max_size]( const std::string& s ) {
                                        return s . size () <= max_size;
                                };
        std::ostream_iterator<std::string> out(std::cout, "\n");
        out = std::copy_if(v.begin(), v.end(), out, corta);
        out = std::copy_if(l.begin(), l.end(), out, corta);
}
```

//////////////////////////

```
template <typename Iter, typename UnaryPred>
Iter find_if(Iter first, Iter last, UnaryPred pred) {
        for ( ; first != last; ++first) {
                if (pred(*first))
                        return first;
        }
        return last;
}

bool pari (long i) { return i % 2 == 0; }

struct Pari{
        //...
};
void foo(const std::vector<long>& vl) {

        auto it = vl.begin();

        for ( ; it != vl.end(); ++it) {
                if (*it % 2 == 0)
                        break;
        }

        auto it = std::find_if(vl.begin, vl.end(), pari); //chiama la funzione bool
pari (...)
        auto it = std::find_if(vl.begin(), vl.end(), Pari{}); //chiama la classe
struct Pari{...}
        auto it = std::find_if(vl.begin(), vl.end(),
                                []long i) { return i % 2 == 0; });

        std::cout<< *it;

        for (const auto& i : vl) { //for range loop, zucchero sintattico, itera su
tutti gli elementi contenuti in vl
                if (i % 2 == 0) {
                        std::cout << i;
                        break;
                }
        }
        //il for range loop funziona anche sugli array
        long al[1000];

        for (const auto& i : al) {
                if (i % 2 == 0) {
                std::cout << i;
                break;
                }
        } //funziona anche senza le funzioni .begin() e .end(), perche tra i vari
algoritmi forniti dalla libreria standard, ci sono anche alcuni funzioni esterne ai
contenitori (e. g. vector) che danno la possibilità di prendere l'inizio della
sequenza memorizzata all'interno del contenitore:
}
```

```
//versione base, begin templatica, prende un typename Cont (contenitore) se passano un contenitore per riferimento (Cont& c)
template <typename Cont>
auto begin(Cont& c) -> decltype(c.begin()) { //decltype operatore che prende un'espressione e torna il tipo di quella espressione
        return c.begin();
}

//stessa cosa per il template .end()
template <typename Cont>
auto end(Cont& c) -> decltype(c.end()) {
        return c.end();
}

//dal momento che gli array non sono classi, mettiamo in overloading un altro template di funzione pensato per funzionare con gli array di n elementi di tipo T
template <typename Cont, std::size_t N>
T* begin( T(&array)[N]){ //array di tipo T, lungo esattamente N, passato per riferimento per evitare il typedecay (arriva il puntatore al primo elemento e non sa più quant'è grande l'array)
        return array;
}
//funzione begin parametrica su due argomenti, il primo è il tipo degli elementi contenuti nell'array, la seconda è una costante nota a tempo di compilazione; restituisce un puntatore ad intero

template <typename Cont, std::size_t N>
T* end( T(&array)[N]) {
        return array + N;
} //puntatore a uno dopo l'ultima posizione dell'array
```

20 - Compilazione dei template

La compilazione del template avviene in due fasi:

1. **definizione del template**: il compilatore analizza il codice del template in modo generico, senza conoscere i tipi o i valori specifici che verranno utilizzati in fase di istanziazione. Può effettuare solo controlli di tipo sintattico e di semantica statica per le parti di codice indipendenti dai parametri del template.
2. **istanziazione del template**: qui il compilatore analizza il template utilizzando i tipi o i valori concreti forniti per i parametri. È in questa fase che il compilatore verifica se tutte le operazioni dipendenti dal tipo sono valide per i tipi specificati.
   nella prima fase il compilatore lavora con informazioni incomplete:

```
template <typename T >
void incr (int& i, T& t ) {
        ++i; // espressione indipendente dai parametri del template dove è possibile effettuare tutti i controlli di sicurezza
        ++t; // espressione dipendente dai parametri del template
}
```

Conseguenze:

- la definizione di un template deve essere disponibile in tutti i punti del programma che richiedono l'istanziazione;
- dal momento che l'istanziazione può avvenire in più unità di traduzione, per non violare il principio DRY (Don't Repeat Yourself) le definizione del template si trova all'interno dell'header file;
- la ODR (One Definition Rule) ammette che le funzioni templatiche siano definite più di una volta (come le funzioni inline) a patto che:
  - si trovino in unità di traduzione diverse,
  - siano definite con la stessa sintassi,
  - siano definite con la stessa semantica.

Esistono tre modi per organizzare i codici sorgente quando si definiscono le classi templatiche:

1. **includere definizioni e dichiarazioni nello stesso file**: includere tutto il codice dei template prima di ogni loro uso in un file header (la definizione completa del template è visibile prima che il compilatore incontri il codice che lo utilizza);
   - utilizzato più frequentemente;
   - "modello di compilazione dei template per inclusione";

```
// File Stack.hh
template <typename T>
class Stack {
  void push(const T& value);
};

template <typename T>
void Stack<T>::push(const T& value) {
  // implementazione
}
```

2. **separare dichiarazioni e definizioni**: le dichiarazioni del template vengono incluse in un file `.hh`, mentre le definizioni vengono inserite in un file `.cpp` separati. È comunque necessario includere il file delle definizioni ogni volta che si usa il template.
   - variante del primo metodo;
   - utilizzato solo se necessario (ad esempio nel caso di funzioni templatiche ricorsive).

```
// File Stack.h (dichiarazioni)
template <typename T>
class Stack {
  void push(const T& value);
};

// File Stack.cpp (definizioni)
template <typename T>
void Stack<T>::push(const T& value) {
  // implementazione
}
```

3. **istanziazioni esplicite** utilizzare le istanziazioni esplicite di template (dichiarare e definire i template in file separati, garantendo che le istanziazioni vengano generate in un'unica unità di traduzione):
   - includere solo le dichiarazioni dei template e le dichiarazioni di istanziazione esplicita prima di ogni loro uso nell'unità di traduzione
   - assicurarsi che le definizioni dei template e le definizioni di istanziazione esplicita siano fornite una sola volta in un'altra unità di traduzione
     Più complicato, meno flessibile, utilizzato per ridurre i tempi di compilazione.

```
// File Stack.h (dichiarazioni)
template <typename T>
class Stack {
  void push(const T& value);
};

// File Stack.cpp (definizioni)
template <typename T>
void Stack<T>::push(const T& value) {
  // implementazione
}

// Istanziazioni esplicite
template class Stack<int>;
template class Stack<double>;
```

Keyword `typename`

Quando un template fa riferimento a un tipo dipendente da un parametro template, il compilatore può confondersi tra un valore e un tipo. Per chiarire che si tratta di un tipo, si utilizza la keyword `typename`.
Problema: esempio non templatico

```
struct S {
  using value_type = int;
};

void foo(const S& s) {
  S::value_type* ptr; // ok
}
```

Il compilatore capisce immediatamente che `S::value_type` è un tipo. Tuttavia, se templatizziamo la classe `S`:

```
template <typename T>
struct S {
  using value_type = T;
};

template <typename T>
void foo(const S<T>& s) {
```

```
  S<T>::value_type* ptr; // errore
}
```

Il compilatore non sa se `S<T>::value_type` è un tipo o un valore, e quindi segnala un errore.
Soluzione: aggiungendo la keyword `typename`, indichiamo esplicitamente al compilatore che si tratta di un tipo:

```
template <typename T>
void foo(const S<T>& s) {
  typename S<T>::value_type* ptr; // ok
}
```

In alcuni casi però il problema può non essere immediatamente evidente:

```
int p = 10;

template <typename T>
void foo(const S<T>& s) {
  S<T>::value_type* p;   // compila senza errori
}
```

Il compilatore interpreta l'istruzione come un'operazione binaria (`S<T>::value_type * p`), dove `p` è la variabile globale dichiarata come intero. Questo comportamento può portare a errori logici difficili da individuare.

22 - Deduzione automatica dei tipi di dato

Template type deduction

E' il processo attraverso cui il compilatore deduce automaticamente i tipi di dati associati ai parametri di un template. Semplifica l'uso dei template e consente di evitare la scrittura della lista di argomenti da associare ai parametri del template di funzione.

- riduce il codice scritto
- previene errori dovuti a specifiche manuali

Regole Generali per la deduzione nei Template

```
template <typename TT>
void f(PT param);
```

- TT: parametro del template
- **PT**: è il tipo dichiarato del parametro `param` della funzione. È definito in funzione di `TT`
  Quando viene chiamata `f(arg)`, il compilatore guarda il tipo di arg e deduce:
- `tt` il tipo dedotto per il parametro template `TT`
- `pt` il tipo dedotto per `PT` in funzione di `tt`
  N.B. i tipi dedotti `tt` e `pt` sono correlati, ma spesso non sono identici
  il processo di deduzione si dividono in tre casi principali:

1. **PT** è sintatticamente uguale a `TT&&`, ovvero è una *universal reference*
2. **PT** è un tipo puntatore o riferimento (ma non una universal reference)
3. **PT** non è né un puntatore né un riferimento

Caso 1: Universal Reference (`PT` è `TT&&`)

- Sono riferimenti speciali che possono essere sia riferimenti a *lvalue* (oggetti già esistenti) che a *rvalue* (oggetti temporanei).
- **Differenza rispetto ad altri riferimenti**:
  - `TT&&` → *universal reference* (deduce sia rvalue che lvalue);
  - `const TT&&` → *rvalue reference* (deduce solo rvalue);
  - `std::vector<TT>&&` → rvalue reference (riferito a un tipo specifico come `std::vector`).
- la **deduzione** dipende dal tipo dell'argomento:
  - se `arg` è un *lvalue*, il tipo dedotto per **PT** sarà un riferimento a *lvalue* (`TT&`);
  - se `arg` è un *rvalue*, il tipo dedotto per **PT** sarà un riferimento a *rvalue* (`TT&&`)
  esempi:

1. rvalue

```
int i = 0;
f(5);   //argomento: rvalue (5) -> int
                //deduzione:
                //TT = int
                //PT = int&& (riferimento a rvalue)
```

2. lvalue

```
int i = 0;
f(i);          // argomento: lvalue (i) -> int&
               // deduzione:
               // TT = int&
               // PT = int& (riferimento a lvalue)
```

3. `const TT&&` non è universal reference, in quanto è *rvalue reference*
4. `std::vector&&` non è universal reference, in quanto è *rvalue reference*
5. std::move

```
f(std::move(i));  // argomento: rvalue (trasformazione di i con std::move) -> int&&
                  // deduzione:
                  // TT = int
                  // PT = int&& (riferimento a rvalue)
```

Vantaggi:

- permettono di scrivere template di funzione che funzionano sia con lvalue sia con rvalue senza duplicare il codice
- supportano perfettamente il "forwarding" (passaggio di argomenti senza perdita di informazioni sul tipo)

Caso 2: PT puntatore

Il compilatore effettua un *pattern matching* per far coincidere il tipo dell'argomento con un puntatore o con un riferimento.

- **Regole di deduzione**:
  - il compilatore deduce il tipo `TT` che "spiega" `PT`
  - se `PT` è un riferimento (es. `TT&` o `const TT&`), il riferimento viene mantenuto;
  - se `PT` è un puntatore (es. `TT*` o `const TT*`), i qualificatori const vengono dedotti.
  Esempi puntatori:

1. puntatore semplice

```
int i = 0;
f(&i);          // argomento: puntatore a int (&i)
                // deduzione:
                // TT = int
                // PT = int* (puntatore a int)
```

2. puntatore a const

```
const int ci = 1;
f(&ci);         // argomento: puntatore a const int (&ci)
                // deduzione:
                // TT = const int
                // PT = const int* (puntatore a const int)
```

3. puntatore con const nel parametro

```
template <typename TT> void f(const TT* param);
f(&i);          // argomento: puntatore a int (&i)
                // deduzione:
                // TT = int
                // PT = const int* (puntatore costante)
```

Esempi riferimenti:

1. riferimento a lvalue

```
int i = 0;
f(i);           // argomento: lvalue
                // deduzione:
                // TT = int
                // PT = int& (riferimento a int)
```

2. riferimento a const lvalue

```
const int ci = 1;
f(ci);          // argomento: const int
                // deduzione:
```

```
                // TT = const int
                // PT = const int& (riferimento costante a int)
```

3. riferimento costante esplicito

```
template <typename TT> void f(const TT& param);
f(i);           // argomento: int
                // deduzione:
                // TT = int
                // PT = const int& (riferimento costante)
```

Caso 3: PT né puntatore né riferimento

Quando il parametro del template viene passato per valore, il compilatore crea una copia dell'elemento passato:

- qualsiasi qualificatore const o riferimento (&, &&) dell'argomento viene rimosso nella deduzione del tipo
- il parametro della funzione è un oggetto separato
  Esempi:

1. oggetto semplice:

```
int i = 0;
f(i);           // argomento: lvalue (int)
                // deduzione:
                // TT = int
                // PT = int (nessun riferimento o costante)
```

2. oggetto costante:

```
const int ci = 1;
f(ci);          // argomento: const int
                // deduzione:
                // TT = int (senza const)
                // PT = int
```

3. qualificatori interni (const all'interno del tipo):

```
const char* const p = "Hello";
f(p);           // argomento: const char* const
                // deduzione:
                // TT = const char* (const interno conservato)
                // PT = const char*
```

Deduzione dei tipi con `auto`

Permette di far dedurre al compilatore il tipo di una variabile basandosi sul suo inizializzatore, inoltre torna utile quando dobbiamo dare un nome a una variabile di una funzione lambda in quanto il tipo non potremmo scriverlo. Esempi:

```
auto i = 5;          // i ha tipo int
const auto d = 5.3;  // d ha tipo const double
auto p = "Hello";    // p ha tipo const char* const
```

Le regole per l'auto type deduction sono simili a quelle dei template. Esempio:
`auto& ri = ci; // ri = const int&`
Inoltre, caso particolare rispetto alla *template type deduction*, con l'utilizzo delle parentesi graffe possiamo ottenere:
`auto i = {1}; // Tipo dedotto: std::initializer_list<int>`
Alcune linee guida di programmazione suggeriscono di usare `auto` quasi sempre, AAA (Almost Always Auto).

Overloading e template di funzione

Nelle regole di risoluzione dell'overloading esistono delle *regole speciali* nel caso in cui troviamo dei template di funzione. N.B. il template non è una funzione, ma una sua specifica istanza sì, infatti quando cerco una funzione candidata per la risoluzione dell'overloading mi riferisco ad un'istanza del template.
Per decidere se una istanza / specializzazione di un template di funzione è un candidato valido dobbiamo verificare che sia possibile effettuare la deduzione dei parametri del template, durante questo processo si applicano solo le corrispondenze esatte. Se il processo di deduzione ha successo l'istanza del template diventa utilizzabile. Le regole specifiche per i template si applicano nella **terza fase**, quando occorre trovare la migliore funzione utilizzabile.

Ordinamento parziale dei template di funzione

I template di funzione con lo stesso nome e visibili nello stesso scope sono ordinati parzialmente rispetto a una "**relazione di specificità**". Definizione:

> **Definizione**
>
> Dato un template primario di funzione *X*, denotiamo con *istanze*(X) l'insieme di tutte le sue possibili istanziazioni.
> Si dice che il template di funzione X è più specifico del template di funzione Y se vale l'inclusione propria $istanze(X) \subset istanze(Y)$.

Esempio:

```
//1: dice solamente che il primo argomento è di tipo T e il secondo di tipo U.
template <typename T, typename U> void foo(T t1, U u2);
//2: dice che il secondo argomento, oltre ad essere di tipo U, è un puntatore. E'
strettamente più specifico del primo.
template <typename T, typename U> void foo(T t1, U* u2);
//3: i parametri sono dello stesso tipo.
template <typename T> void foo(T t1, T t2);

//i template 2 e 3 non sono confrontabili
void foo(int, int*) //`e istanza di 2 ma non di 3
void foo(int, int) //`e istanza di 3 ma non di 2
```

Regole Aggiuntive

Quando si cerca tra le funzioni utilizzabili, la funzione migliore (se esiste), consideriamo:

- preferenza per i **template più specifici**
- preferenza per le **funzioni non templatiche**: se dopo aver eliminato le istanze meno specifiche si ottiene una situazione di ambiguità, allora si eliminano dalle istanze tutte le istanze di template.

```
//si considerino le seguenti dichiarazioni:
template <typename T, typename U> //#1
void foo( T a1, U a2);

template <typename T, typename U> //#2
void foo(T a1, U* a2);

template <typename T> //#3
void foo(T a1, T a2);

void foo(int* a1, int* a2); //#4

//chiamata
foo(42, 42);
//#2 e #4 non sono utilizzabili
//la funzione migliore è quella che si ottiene istanziando #3, preferito rispetto a
#1 per specificità
void foo<int>(int, int)

//chiamata
foo(&i, &i) //i di tipo int
//sono tutte utilizzabili e la #1 è scartata per specificità
//ci sarebbe ambiguità tra istanza #2
void foo<int*, int>(int*, int*)
//e istanza #3
void foo<int*>(int*, int*)
//e la funzione #4
//=> la migliore è la quattro (preferenza per non templatica)
```

26 - Classi dinamiche

Classi derivate e relazione "IS-A"

La relazione "IS-A" rappresenta uno dei principi fondamentali dell'ereditarietà in c++. Si verifica quando una classe derivata può essere considerata come una specializzazione della classe base.

```
class Base {
public:
    virtual void print() const { std::cout << "Base\n"; }
};

class Derived : public Base {
public:
    void print() const override { std::cout << "Derived\n"; }
```

```
};

int main() {
    Derived d;
    Base* base_ptr = &d; // Up-Cast implicito
    base_ptr->print();   // Output: Derived (se il metodo è virtuale)
}
```

In questo caso, la relazione "IS-A" si traduce nel fatto che `Derived` è una specializzazione di `Base`, quindi un oggetto `Derived` può essere usato ovunque sia richiesto un oggetto `Base`.

Derivazione non pubblica

Se la derivazione è **private** o **protected**, l'up-cast implicito non è consentito al di fuori del contesto della classe derivata o di una funzione `friend`. Questo serve a nascondere il fatto che una classe deriva da un'altra.

```
class Base {};

class Derived : private Base {}; // Derivazione privata

int main() {
    Derived d;
    Base* base_ptr = &d; // Errore: conversione non consentita
    return 0;
}
```

Metodi virtuali e classi dinamiche

I **metodi virtuali** consentono di implementare il **polimorfismo runtime**, una caratteristica fondamentale di C++. Grazie ai metodi virtuali, la risoluzione delle chiamate a una funzione membro non avviene a tempo di compilazione, ma a tempo di esecuzione, in base al tipo dinamico dell'oggetto.
Quando una funzione membro viene dichiarata `virtual` nella classe base, le sue ridefinizioni nelle classi derivate vengono chiamate tramite un meccanismo di dispatch dinamico. Questo comportamento viene gestito utilizzando una struttura chiamata **vtable** (tabella virtuale).

```
class Base {
public:
    virtual void print() const {
        std::cout << "Base\n";
    }
};

class Derived : public Base {
public:
    void print() const override {
        std::cout << "Derived\n";
    }
};

int main() {
    Base* ptr = new Derived();
```

```
    ptr->print(); // Output: Derived
    delete ptr;
}
```

In questo esempio:

1. la classe `Base` è una **classe dinamica** perché contiene un metodo virtuale;
2. quando `ptr->print()` viene chiamato, il compilatore non sa a quale versione del metodo `print` fare riferimento. Il **runtime** interroga la vtable per scoprire quale metodo chiamare in base al tipo dinamico dell'oggetto (`Derived`).
   In particolare:

```
Base* ptr = new Derived;
```

- il **tipo statico** di `ptr` è `Base*` perché `ptr` è dichiarato come puntatore a `Base` :
  - **tipo statico** := tipo che il compilatore conosce al momento della compilazione.
- il **tipo dinamico** di `*ptr` è `Derived` , perché `ptr` punta effettivamente a un oggetto della classe `Derived` :
  - **tipo dinamico** := tipo effettivo dell'oggetto a cui il puntatore (o riferimento) punta durante l'esecuzione.

Metodi virtuali puri e classi astratte

Metodi virtuali puri:

è un metodo dichiarato nella forma:

```
virtual void metodo() = 0;
```

Un metodo virtuale puro:

- non ha implementazione nella classe base;
- obbliga le classi derivate a fornire una propria implementazione.
  Le classi che contengono almeno un metodo virtuale puro sono dette **classi astratte** e non possono essere istanziate.

```
class Astratta {
public:
    virtual void metodo_puro() = 0; // Metodo virtuale puro
};

class Concreta : public Astratta {
public:
    void metodo_puro() override {
        std::cout << "Implementazione concreta\n";
    }
};

int main() {
    Astratta a; // Errore: la classe è astratta
```

```
    Concreta c;
    c.metodo_puro(); // Output: Implementazione concreta
}
```

Distruttori virtuali

Un distruttore virtuale è essenziale per garantire che la distruzione degli oggetti derivate avvenga correttamente quando si usa un puntatore alla classe base.
Se un distruttore non è virtuale, quando un oggetto derivato viene distrutto tramite un puntatore alla classe base, viene chiamato solo il distruttore della classe base, causando **memory leak**.
Esempio senza distruttore virtuale:

```
class Base {
public:
    ~Base() { std::cout << "Distruttore Base\n"; }
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Distruttore Derived\n"; }
};

int main() {
    Base* b = new Derived;
    delete b; // Output: Distruttore Base (ma non Distruttore Derived!)
}
```

Per evitare questo, il distruttore deve essere dichiarato `virtual` nella classe base:

```
class Base {
public:
    virtual ~Base() { std::cout << "Distruttore Base\n"; }
};
```

Risoluzione dell'overriding

Perché l'overriding funzioni correttamente, devono essere rispettate alcune condizioni:

1. **il metodo deve essere dichiarato virtuale** nella classe base;
2. **deve essere invocato tramite un puntatore o un riferimento** alla classe base;
3. **la classe derivata deve fornire un'implementazione** del metodo;
4. **non deve esserci qualificazione esplicita**: se si usa un qualificatore, si chiama esplicitamente il metodo della classe qualificata.
   Esempio di qualificazione esplicita:

```
class Base {
public:
    virtual void print() const {
        std::cout << "Base\n";
    }
};
```

```cpp
class Derived : public Base {
public:
    void print() const override {
        std::cout << "Derived\n";
    }
};

int main() {
    Derived d;
    d.print();           // Output: Derived
    d.Base::print();     // Output: Base (qualificazione esplicita)
}
```

27 - Cast

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"
   I cast devono essere utilizzati solo quando necessario. È possibile classificare gli usi dei cast in base alla motivazione:

Caso 1: Implementazione di conversioni non consentite implicitamente

Il cast implementa una conversione di tipo non consentita dal linguaggio come conversione implicita, poiché potrebbe essere una frequente fonte di errori. Utilizzando un cast esplicito, il programmatore si assume la responsabilità della correttezza della conversione.

```cpp
struct B { /* ... */ };
struct D : public B { /* ... */ };
D d;
B* b_ptr = &d;
// b_ptr è un puntatore a B, ma dinamicamente punta a un oggetto di tipo D.
D* d_ptr = static_cast<D*>(b_ptr); // Il programmatore forza il down-cast.
```

N.B. se qualcuno nel frattempo avesse modificato `b_ptr` e questo non puntasse più ad un oggetto di tipo D, si ottiene un *undefined behavior*.

Caso 2: Uso di `dynamic_cast` per verificare la conversione a runtime

Il programmatore utilizza un cast dinamico per verificare a runtime se la conversione è consentita.

```cpp
struct B { virtual ~B() {} }; // Classe base dinamica
struct D : public B { /* ... */ };
```

```
void foo(B* b_ptr) {
    if (D* d_ptr = dynamic_cast<D*>(b_ptr)) {
        // Uso d_ptr perché il cast è andato a buon fine.
    } else {
        // b_ptr non punta a un oggetto di tipo D.
    }
}
```

Caso 3: Conversioni consentite implicitamente, ma fatte per documentazione

Il cast non è strettamente necessario (la conversione implicita è permessa), ma il programmatore preferisce renderlo esplicito per migliorare la leggibilità e documentare l'intenzione.

```
double d = 3.14;
int approx = static_cast<int>(d); // Evidenzia che si perde informazione.
```

Caso 4: Conversione a `void` per ignorare un valore

In alcuni casi, si usa un cast esplicito a `void` per ignorare il valore di un'espressione e silenziare warning del compilatore.

```
void foo(int pos, int size) {
    assert(0 <= pos && pos < size);
    static_cast<void>(size); // Silenzia il warning sul mancato uso di size.
}
/// in casi come questo si tollera il cast in stile C:
(void)size;
```

Tipologie di cast

1. `static_cast`

È il cast più comune. Si usa per conversioni ben definite e controllate. La sintassi è:

```
static_cast<T>(expr)
```

Esempi:

- conversioni implicite:

  ```
  double d = 3.14;
  int i = static_cast<int>(d);
  ```

- downcast nelle gerarchie di classi :

  ```
  B* b_ptr = new D();
  D* d_ptr = static_cast<D*>(b_ptr);
  ```

- conversioni a `void` :

```
static_cast<void>(expr);
```

2. `dynamic_cast`

Si usa per verificare la validità di conversioni in gerarchie di classi a runtime. Funziona solo con classi dinamiche (contenenti almeno un metodo virtuale).

```
struct B { virtual ~B() {} };
struct D : public B { /* ... */ };

B* b_ptr = new D();
D* d_ptr = dynamic_cast<D*>(b_ptr); // Controlla se b_ptr è effettivamente un D.
if (d_ptr) {
    // Il cast è valido.
} else {
    // Il cast è fallito.
}
```

3. `const_cast`

Serve per rimuovere (o aggiungere) la qualificazione `const`.

```
void modifica(const int& ci) {
    int& i = const_cast<int&>(ci);
    i = 42; // Modifica il valore ignorando la promessa di non modificarlo.
}
```

Nota: L'abuso di `const_cast` è considerato cattivo stile. In casi legittimi, può essere usato per aggiornare dati interni dichiarati `mutable`.

4. `reinterpret_cast`

Il cast più pericoloso. Si usa per reinterpretare un bit pattern senza alcuna garanzia di validità.

```
int x = 42;
void* v_ptr = &x;
int* i_ptr = reinterpret_cast<int*>(v_ptr);
```

Nota: è responsabilità del programmatore assicurarsi che il risultato sia valido.

5. Cast Funzionale

Ha la sintassi:

```
T(expr)
```

Si usa spesso per tipi primitivi o per costruire oggetti.

```
int i = int(3.14); //cast funzionale
```

6. Cast stile C

Ha la sintassi:

```
(T)expr
```

È considerato cattivo stile, eccetto quando viene usato per conversioni a `void`.

28 - Principi SOLID

Con l'acrostico SOLID si identificano i cosiddetti "primi 5 principi" della progettazione object oriented. Lo scopo di essi è fornire una guida verso lo sviluppo di progetti che siano più "flessibili", ovvero progetti per i quali sia relativamente semplificato effettuare modifiche in termini di:

- manutenzione delle funzionalità esistenti;
- estensione delle funzionalità supportate.
  S ⇒ SRP (Single Responsibility Principle)
  O ⇒ OCP (Open-Closed Principle)
  L ⇒ LSP (Liskov Substitution Principle)
  I ⇒ ISP (Interface Segregation Principle)
  D ⇒ DIP (Dependency Inversion Principle)
  Parliamo di principi e non di tecniche o metodi in quanto non sono immediatamente applicabili, la loro applicazione richiede di valutare se sia o meno opportuno applicarli nei vari contesti concreti che si presentano.
  Quando un software cresce in termini di complessità, aumenta il rischio di creare codice fragile o difficile da mantenere. Alcune problematiche che SOLID aiuta a prevenire sono:
- **accoppiamento eccessivo**: le classi dipendono troppo l'una dall'altra, rendendo difficile apportare modifiche senza causare effetti collaterali;
- **ridondanza**: il codice viene duplicato perché manca modularità e riutilizzo;
- **rigidità**: è difficile aggiungere nuove funzionalità perché il codice esistente deve essere modificato in modo esteso;
- **opacità**: il codice diventa complesso e difficile da leggere.
  I principi SOLID favoriscono lo sviluppo di software scalabile e flessibile, capace di adattarsi facilmente ai cambiamenti richiesti nel tempo.

SRP - Single Responsibility Principle

(Principio della Responsabilità Unica)
Una classe, funzione o modulo dovrebbe essere **responsabile di un'unica cosa**. Si può dire anche che, ogni classe dovrebbe avere **un solo motivo per essere modificata**: se esistono più motivi distinti vuol dire che la classe si assume più responsabilità e quindi dovrebbe essere suddivisa in più componenti.
Quando una classe ha più responsabilità:

- diventa più difficile da mantenere;
- qualsiasi modifica a una delle responsabilità potrebbe causare effetti collaterali su altre funzionalità;
- diventa meno riutilizzabile perché accoppia logiche diverse.

Il rispetto di questo principio porta a codice:

- più manutenibile;
- più facile da riutilizzare.
  Esempio: consideriamo una classe `Report` che:

1. genera un report;
2. lo salva su disco;
3. lo invia tramite email.
   Questa classe viola il principio SRP perché ha tre responsabilità distinte.
   Correzione:

- `ReportGenerator` → crea il report;
- `ReportSaver` → salva il report;
- `EmailSender` → invia il report.

Una classe che deve manipolare più risorse (in maniera exception safe) non deve prendersi carico anche della corretta gestione dell'acquisizione e rilascio; piuttosto dovrebbe delegare questi compiti ad altre classi gestore.

OCP - Open-Closed Principle

(Principio Aperto-Chiuso)
Le entità software (classi, moduli, funzioni) dovrebbero essere **aperte per estensioni**, ma **chiuse per modifiche**.
In altre parole, un software ben progettato deve rendere semplice l'aggiunta di funzionalità, senza che per fare ciò sia necessario modificare il codice esistente, per evitare di introdurre bug o effetti collaterali.
Per ottenere un progetto coerente con il principio OCP occorre individuare le parti del software che, probabilmente, saranno oggetto di modifiche in futuro e applicare ad esse opportuni costrutti di astrazione.
Esempio: confrontando le due varianti (`old_style` vs `oo_style`) del progetto "fattoria" le parti di codice che saranno oggetto di cambiamento è l'introduzione di nuovi animali, che dovranno essere utilizzabili nel codice utente. Notiamo che:

- `old_style` :
  - ha un'unica classe `Animale` che implementa tutti gli animali concreti;
  - l'aggiunta di nuovi animali, infatti, comporta la modifica della classe;
  - il codice è aperto alle estensioni, ma solo parzialmente chiuso alle modifiche;
    Pur non dovendo modificare il codice che genera le strofe, ogni estensione che aggiunge un animale rischia di rompere il codice preesistente.
- `oo_style` :
  - abbiamo una classe astratta `Animale` che fornisce l'interfaccia, senza dettagli implementativi;
  - il codice è aperto alle estensioni, in quanto per aggiungere un nuovo animale, basta creare una nuova classe che implementa (con derivazione pubblica "IS-A") l'interfaccia astratta;
  - il codice è chiuso alle modifiche: le aggiunte non hanno impatto sul codice che genera la strofa e nemmeno sulle classi degli altri animali.
    N.B. in entrambi i casi l'aggiunta di nuovi animali prevede modifiche al modulo `Maker.cc` , per consentire ai nuovi animali di essere utilizzati dal programma. Quindi la "chiusura alle modifiche" non può mai essere totale.

DIP - Dependency Inversion Principle

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello; entrambi dovrebbero dipendere da astrazioni. Le astrazioni non devono dipendere dai dettagli, ma i dettagli devono dipendere dalle astrazioni.
In altre parole, invece di far dipendere il codice da implementazioni concrete, si dovrebbe basare su astrazioni, come interfacce o classi astratte. Questo principio fa una classifica delle dipendenze tra moduli software, stabilendo che alcune sono ammesse (inevitabile e innocue), mentre altre sono da evitare (dannose).

- *dipendenze buone*: quelle verso i concetti astratti;
- *dipendenze cattive*: quelle verso i dettagli implementativi.
  Spesso il software viene progettato con un approccio *top-down*: partendo dal problema generale, lo si suddivide in sottoproblemi via via sempre più piccoli. Si arriva cosi a una stratificazione del codice, dove i moduli a livello più alto usano (quindi dipendono da) i moduli a livello più basso, creando cosi delle dipendenze (dall'astratto al concreto).
  Il principio DIP suggerisce di **invertire** le dipendenze sostituendolo con altre che non creano problemi, per farlo occorre individuare le **interfacce astratte**, che non dipendono da dettagli implementativi:
- i moduli ad alto livello vengono modificati per usare le interfacce astratte;
- i moduli a basso livello vengono modificati per implementare le interfacce astratte.
  In questo modo miglioriamo anche l'aderenza al principio OCP, in quanto adesso è possibile estendere il software senza influenzare i moduli ad alto livello.

LSP - Liskov Substitution Principle

(Principio di Sostituibilità di Liskov)
Gli oggetti di una classe derivata devono poter sostituire quelli della classe base senza alterare il comportamento corretto del programma.

In altre parole, il comportamento di una classe derivata deve essere coerente con quello della classe base. Le classi derivate non devono violare le aspettative degli utenti della classe base.
Il principio LSP definisce il cosiddetto **behavioral subtyping**: siccome la classe derivata dichiara di essere in relazione IS-A rispetto alla classe base, gli oggetti della classe derivata non solo devono fornire (sintatticamente) i metodi dichiarati dalla classe base, ma si devono anche comportare (aspetto semantico) come se fossero degli oggetti della classe base.
Le corrispondenze di comportamento tra le due classi non deve essere intesa in senso assoluto, ma limitata alle **aspettative legittime** che può avere un utente della classe base.
Le aspettative legittime sono quelle stabilite dai *contratti* (invarianti di classe, precondizioni e postcondizioni dei metodi) che la classe base ha stipulato con i suoi utenti.
Infatti, il principio LSP (e il behavioral subtyping) sono quindi in connessione stretta con la **programmazione per contratto**: quando la classe derivata dichiara di essere in relazione IS-A con la classe base, di fatto si impegna a rispettare tutti i contratti che la classe base 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 Animale, rispetta il contratto stabilito da questa con i suoi utenti.
Un altro esempio: partiamo dalla classe `Rettangolo` e cerchiamo di ottenere la classe `Quadrato`

```
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("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 lung * larg; }
}; //class Rettangolo
```

Per creare la classe `Quadrato`, si potrebbe pensare di implementare la classe usando l'ereditarietà pubblica e il polimorfismo dinamico, dunque le uniche modifiche da fare sono:

- dichiarare i metodi di rettangolo come virtual;
- dichiarare il distruttore di rettangolo virtuale
- derivare pubblicamente `Quadrato` da `Rettangolo`
- fare l'override dei metodi `set_larghezza` e `set_lunghezza`, per assicurarsi che quando si modifica una dimensione sia modificata anche l'altra, cosi da mantenere l'invariante della classe `Quadrato`
  Modifichiamo `Rettangolo`:

```
class Rettangolo {
        ///
        virtual bool check_inv() const { ... }
        virtual long get_lunghezza() const { ... }
        virtual long get_larghezza() const { ... }
        virtual long set_lunghezza(long value) { ... }
        virtual long set_larghezza(long value) { ... }
        virtual long get_area() const { ... }
```

```
        virtual ~Rettangolo() = default;
}; //class Rettangolo

class Quadrato : public Rettangolo {
public:
        Quadrato(long lato) : Rettangolo(lato, lato) {}
        bool check_inv() const override {
                return lung > 0 && lung == larg;
        }
        void set_lunghezza(long value) override {
                Rettangolo::set_lunghezza(value);
                Rettangolo::set_larghezza(value);
        }
        void set_larghezza(long value) override {
                set_lunghezza(value);
        }
}; //class Quadrato
```

Questo progetto viola la LSP, in quanto è falso che qualunque funzione che usa puntatori (o riferimenti) alla classe base Rettangolo possiamo passare invece puntatori (o riferimenti) alla classe derivata Quadrato e soddisfare le legittime aspettative dell'utente. Hanno la stessa interfaccia (aspetto sintattico) ma non sono equivalenti in quanto si comportano diversamente. Ad esempio, la classe Quadrato viola il contratto del metodo `Rettangolo::set_lunghezza`, in quanto nelle postcondizioni stabilisce che va a modificare solo la lunghezza, mentre l'overriding definito da Quadrato va a modificare anche la larghezza. Possiamo allora dire che la relazione Quadrato IS-A Rettangolo non è valida, in quanto un quadrato non si comporta come un rettangolo.

ISP - Interface Segregation Principle

(Principio di Segregazione delle Interfacce)
Un'interfaccia non dovrebbe costringere le classi a implementare metodi che non usano. Interfacce troppo grandi rendono le classi rigide e accoppiate. Cambiare un metodo inutilizzato in un'interfaccia può richiedere modifiche a tutte le classi che la implementano. In altre parole, si dovrebbe preferire tante interfacce piccole (*thin interfaces*) rispetto a poche interfacce grandi (*thick interfaces*). Può essere interpretato come una forma particolare del principio SRP, che si concentra sul caso specifico della progettazione delle interfacce.
L'applicazione di questo principio presuppone la possibilità di utilizzare l'ereditarietà multipla (di interfaccia).

29 - Ereditarietà

Nel caso di polimorfismo dinamico, le classi astratte sono formate tipicamente da:

- metodi virtuali puri;
- il distruttore della classe dichiarato virtuale (ma non puro).
  In alcuni casi occorre complicare il progetto (es. usando l'ereditarietà multipla), nonostante si corra il rischio di fare errori.

Metodi che NON possono essere dichiarati virtuali**

- **Costruttori**:
  I costruttori **non possono essere dichiarati virtuali**. Questo perché, durante la costruzione di un oggetto, il meccanismo di polimorfismo dinamico non è ancora operativo (la vtable non è ancora completamente inizializzata). Per ottenere un comportamento "simile" al polimorfismo, si usa il design pattern *factory method* o l'ereditarietà virtuale per gestire la costruzione di oggetti attraverso puntatori o riferimenti a classi base.
- **Distruttori**:
  I distruttori **possono essere dichiarati virtuali** e spesso **devono esserlo** quando si lavora con polimorfismo dinamico. Questo garantisce che, quando si elimina un oggetto tramite un puntatore alla classe base, il distruttore della classe derivata venga chiamato correttamente, evitando problemi di *memory leak*.
  Esempio:

```
class Base {
public:
    virtual ~Base() {}
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Distruzione di Derived\n"; }
};
```

- **Funzioni membro (non statiche)**:
  Queste **possono essere dichiarate virtuali**, ed è proprio il contesto principale del polimorfismo dinamico. Solo le funzioni membro non statiche possono accedere al puntatore implicito `this`, che è essenziale per il comportamento polimorfico.
- **Funzioni membro statiche**:
  Le funzioni statiche **non possono essere dichiarate virtuali**. Questo perché non fanno parte dell'istanza dell'oggetto e non accedono al puntatore `this`. Di conseguenza, non possono partecipare alla risoluzione dinamica dei metodi.
- **Template di funzioni membro (non statiche)**:
  Le funzioni membro template **possono essere dichiarate virtuali**, ma raramente ha senso farlo. Ogni istanza del template genera un metodo separato per il tipo specifico, il che può rendere difficile mantenere un comportamento polimorfico coerente.
- **Funzioni membro di classi templatiche (non statiche)**:
  Le funzioni membro di classi templatiche **possono essere virtuali**, e ciò può essere utile quando si vuole implementare una gerarchia di classi basate su template.
  Esempio:

```
template <typename T>
class Base {
public:
    virtual void foo() = 0;
};

class Derived : public Base<int> {
public:
```

```
    void foo() override { std::cout << "Derived\n"; }
};
```

Come costruire una copia di un oggetto concreto dato un puntatore/riferimento alla classe base

Per creare una copia di un oggetto concreto dato un puntatore o un riferimento alla classe base, si può utilizzare il **metodo virtuale** `clone`. Questo approccio è comune nei pattern di progettazione, come il *Prototype pattern*.

```
class Base {
public:
    virtual ~Base() {}
    virtual Base* clone() const = 0;
};

class Derived : public Base {
public:
    Derived* clone() const override { return new Derived(*this); }
};
```

In questo modo, chiamando `clone()` su un oggetto tramite un puntatore alla base, si ottiene una copia corretta dell'oggetto derivato.

Invocare un metodo virtuale durante costruzione/distruzione

Invocare un metodo virtuale durante la costruzione o la distruzione di un oggetto può portare a comportamenti inaspettati. Durante queste fasi, il tipo dinamico dell'oggetto è considerato il tipo della classe in corso di costruzione o distruzione, non quello effettivo del derivato.

```
class Base {
public:
    Base() { foo(); }
    virtual void foo() { std::cout << "Base::foo\n"; }
    virtual ~Base() { foo(); }
};

class Derived : public Base {
public:
    void foo() override { std::cout << "Derived::foo\n"; }
};

Derived d;
```

Output:

```
Base::foo
Derived::foo
```

Durante il costruttore, il metodo `foo` della classe `Base` viene chiamato perché il costruttore della classe derivata non è ancora stato eseguito. Durante il distruttore, succede il contrario.

Ereditarietà multipla con classi non astratte

L'ereditarietà multipla introduce complessità che devono essere gestite con attenzione.

Scope e ambiguità

Se due classi base contengono metodi con lo stesso nome, si può incorrere in ambiguità. È necessario specificare la classe di origine:

```
class A {
public:
    void foo() { std::cout << "A\n"; }
};

class B {
public:
    void foo() { std::cout << "B\n"; }
};

class Derived : public A, public B {};

Derived d;
d.foo(); // Errore: ambiguità
d.A::foo(); // Corretto
```

Classi base ripetute

Se una classe è ereditata più volte attraverso diverse catene di ereditarietà, si hanno copie multiple di quella classe base, causando ridondanza.

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ }; // D contiene due istanze di A
```

Classi base virtuali

Per evitare duplicazione di classi base, si utilizza l'ereditarietà virtuale:

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public B, public C { /* ... */ }; // Una sola istanza di A
```

Semantica speciale di inizializzazione

Con classi base virtuali, la classe derivata più specifica è responsabile dell'inizializzazione della classe base comune:

```cpp
class A {
public:
    A(int x) { std::cout << "A: " << x << '\n'; }
};

class B : virtual public A {
public:
    B() : A(0) {} // Non inizializza A
};

class C : virtual public A {
public:
    C() : A(0) {} // Non inizializza A
};

class D : public B, public C {
public:
    D() : A(42), B(), C() {} // A viene inizializzata qui
};
```

Usi del polimorfismo dinamico nella libreria standard

Classi eccezione standard

Il polimorfismo dinamico è alla base del sistema di gestione delle eccezioni in C++. Tutte le eccezioni derivano da `std::exception`, che fornisce il metodo virtuale `what()`:

```cpp
try {
    throw std::runtime_error("Errore!");
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Classi iostream

Le classi della gerarchia `iostream` utilizzano il polimorfismo dinamico per fornire interfacce comuni a flussi di input (`istream`), output (`ostream`), e combinati (`iostream`).

```cpp
std::ostream& stream = std::cout;
stream << "Polimorfismo dinamico!\n";
```