Metodologie di Programmazione

Aldo Tragni

Per altre dispense, visita: https://goo.gl/jB0xc4

Indice

Programma: non è detto che termini (se termina con qualsiasi dato allora si può definire algoritmo).
main(): può anche non contenere dati.
int main(): ritorna un intero per indicare che se:
- restituisce 0: ha fatto ciò che doveva fare.
- altrimenti restituisce un altro numero (errore).
Se nel Main non metto return, funziona (possibilità data solo dal main il quale ha return di default).
return: è una keyword / parola chiave.
cout: oggetto di tipo ostream
<< , :: , ; , () operatori.
int: se inizia con 'O' si considera ottale; con 'Ox' si considera esadecimale. 'Oul' unsigned long.
'0.0' '0.' '.0': sono numeri con la virgola di tipo double.
'0.0f ': numeri con la virgola di tipo float.
Interi: short, int, long, long long, signed e relativi unsigned, unsigned char, char
char è un intero (piccolo) speciale perché stampando si ottiene un carattere.
endl: svuota il buffer (cache): è una funzione.
/n: non svuota il buffer.
Libreria: pacchetto che contiene funzioni già pronte.

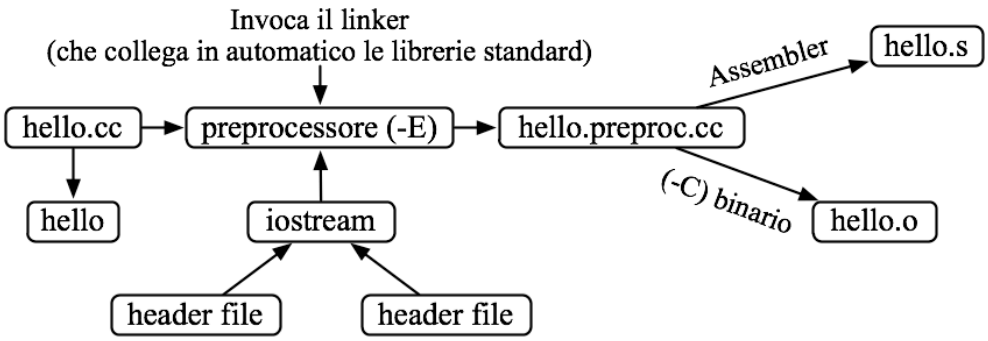
#include <iostream>: è parte del codice sorgente (header file), non è una libreria.
Senza <iostream> ottengo un errore di mancata dichiarazione del cout.
Header file: Un header file (o file di intestazione) è un file che aiuta il programmatore nell'utilizzo di librerie durante la programmazione. Un header file è un semplice file di testo che contiene i prototipi delle funzioni definite nel relativo file cc. I prototipi permettono al compilatore di produrre un codice oggetto che può essere facilmente unito con quello della libreria in futuro, anche senza avere la libreria sottomano al momento.
- **Dichiarazione**: con essa si indica che esiste un'entità con un certo nome (non forniamo un'implementazione particolare).
- **Definizione**: si fornisce anche l'implementazione.

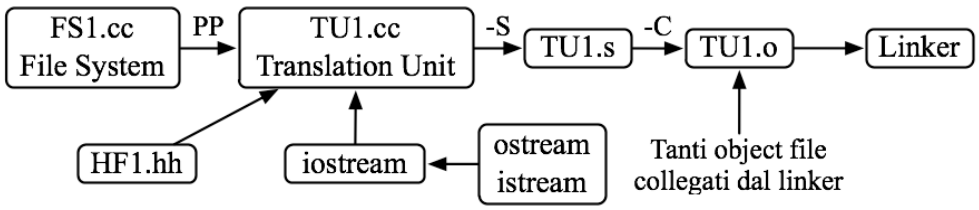
int a=0; // è una **inizializzazione** quindi viene **definita** e non dichiarata.
extern int a; //fornisce una **dichiarazione** di variabile senza inizializzarla (grazie all'extern).
extern int a=0; //**definizione**
extern std::ostream cout; //write one - in questo modo la modifico una sola volta nel header file e si modifica ovunque.
Anche l'operatore << va ridefinito per usare cout e endl.

G++: compilatore
Opzioni:
- W: warming (attiva) ti dice se c'è qualche errore ("massimo livello").
- Wall: controlla "tutti" gli errori.
- Wextra: controlla errori extra.
- O: dove salva l'output.
- E: restituisce il numero di caratteri di "hello.preproc.cc", il file creato dal processore prima di usare il compilatore.
"hello.preroce.cc" contiene tutto (anche gli #include).
- LDD: dice quali librerie invoca il programma.
- Wcin e Wcout: lavorano su carattere più estesi (con più possibili simboli).

Codici:
"G++ -Wall - Wextra hello.cc -O hello"
Esegue il programma indicando eventuali warming
"(G++ -Wall - Wextra hello.cc -O hello) && echo OK"
Se stampa OK è giusto quindi il programma funziona
"G++ -S -Wall - Wextra hello.cc -O hello.s"
Fa una traduzione in assembler (in codice oggetto).
In codice oggetto tutti i nomi vengono resi unici (**name engley**).
"G++ -C -Wall - Wextra hello.cc -O hello.o"
Genera il codice oggetto in binario.
"G++ -E -Wall - Wextra hello.cc -O hello.preproc.cc"
Restituisce il numero di caratteri di "hello.preproc.cc", il file creato dal processore prima di usare il compilatore.

Preprocessore: svolge funzioni prima del compilatore.
Unità di traduzione: è il file che il compilatore compila.
- Ci sono tante unità di traduzione e su ognuna si applica la compilazione.





ERRORI
- Errori sintattici: (soggetto -> verbo -> compilazione oggetto) se non si segue questo schema si sbaglia.
- Errori di semantica statica: sommare intero e stringa.
- Errori di semantica dinamica: provo a leggere un indirizzo di un array che però non si a dove punta.
- Errori grammaticali: usare parole che non esistono nel linguaggio.

Undefined Behavior (UB): problema non risolto ed è colpa tua.
unsigned namespace std; evita alcune ambiguità fra i nomi.
using std::cout; , using std::endl; prende dalla libreria std solo i nomi che servono (posso usare cout senza ambiguità).

- Meno “roba si tira dentro” più nomi posso usare per variabili.
- Ma usare tanti using rende il codice meno riutilizzabile.
Le identità di std sono: cout, endl, <<, ...
- << però non ha mai davanti std::

Namespace: una collezione di nomi di entità, definite dal programmatore, omogeneamente usate in uno o più file sorgente.
Lo scopo dei namespace è quello di evitare confusione ed equivoci nel caso siano necessarie molte entità con nomi simili, fornendo il modo di raggruppare i nomi per categorie.

- Errore di non dichiarazione
Se si cerca di utilizzare una variabile fuori dallo scope della variabile stessa.
Scope: campo di azione (di una variabile).
- Errore di non definizione

```
extern int a;
cout << a << endl;  // errore perché non so cosa stampare
```

"G++ -Wall - Wextra -nostdlib hello.o a.o -O hello"
- nostdlib: non include la libreria std
- hello.o a.o: collego più file oggetto.
Se in hello.o e a.o definisco a, mi darà un errore di doppia definizione.

#include<iostream>: cerca tra gli header file del sistema
#include "a.hh": cerca nella directory stessa il file a.hh
Le entità globali è meglio metterle in un file che poi verrà incluso (a.hh).

Segmentation fault: Si verifica quando un programma tenta di accedere ad una posizione di memoria alla quale non gli è permesso accedere, oppure quando tenta di accedervi in una maniera che non gli è concessa.
Core dumped: caduta improvvisa del programma. Si verifica ad esempio, se si è riempito lo stack.

Continuando a diminuire un numero intero si arriva al minimo numero intero dopo il quale si passa al valore intero più grande.
- Errore di implementazione: i valori di un intero sono implementati.
Per i numeri senza segno, superando il massimo si ottiene lo 0.

```
unsigned long long
fatt(unsigned...)
```
- meglio andare a capo.

- se una funzione richiede un unsigned int nulla vieta che l'utente la richiami con ad es -1

```
unsigned fact (unsigned n) {
  if (n==0U) …
}
```
- unsigned controlla che il valore sia zero unsigned (se l'utente passa -1 non funziona)
- posso anche indicare con un commento quali sono i valori accettati

Controllare gli errori

Modi:
- **Variabile globale**: sempre a zero che cambia in base all'errore.

Problema: ogni volta che si richiama una funzione, c'è da controllare l'errore.
- **Modificare la variabile per generare un errore**.

Problema: anche qui da controllare ogni volta. Si potrebbero avere dei valori che hanno senso ora ma magari in futuro no.
- **Try catch**
- **Asserzioni (assert)**

Try Catch

Eccezioni: di default non potete ignorarle (anche se ci si dimentica, si propagano finchè tale errore non viene gestito).
- Non sporca il risultato.

```
if (n<0)
throw 123; //Usando ad esempio codici numerici di errore specificati in un documento a parte.
```

- Il throw però costa quindi scarico la responsabilità sull'utente scrivendo in una documentazione cosa necessita la funzione per funzionare.

Compromesso tra gestione degli errori mia o dell'utente:

Assert

Funzioni che restituiscono vero o falso in modo da sapere la riga di errore (assert (n>=0);).
Per usare assert serve #include<cassert>.
Durante il debug quindi verrà usato assert (controllo) per trovare l'errore.

"G++ -Wall - Wextra -dndebut hello.cc a.cc -O hello"
- dndebug: non fa il debug
- se non si fa il debug gli assert non vengono effettuati.

Sapere il valore massimo rappresentabile (in C++):
Massimo per un tipo:
- numeric_limits<char>::max();
- CHAR_MAX
- SCHAR_MAX // S -> signed
- UCHAR_MAX // U -> unsigned

Minimo per un tipo:
- numeric_limits<char>::min();
- CHAR_MIN
- SCHAR_ MIN // S -> signed
- UCHAR_ MIN // U -> unsigned

Modo per avere numeri lunghi a piacere (in C++):
E' necessario includere una libreria che gestisca tali numeri.

ESEMPI:

File: a.hh

```
#include <gmpxx.h>
extern int a;

//deve essere invocato con n>=0
mpz_class fact(mpz_class n);
```

File: hello

```
#include "a.hh"
#include <limits>
#include <iostream>

int a;  //SE globale, di default è definita = 0; con “explicit” non sarebbe definita
void funz(...) {
  int a;  //SE non globale, non è definita di default
  auto b = 3.2;  //determina in automatico il tipo della variabile
  auto int a;  //nel c++ 11 non si può più fare; nelle vecchie versioni, definiva “a” di dafault a 0
}

int main() {
std::cout << "hello" << std::endl;
std::cout << a << std::endl;

std::cout << sizeof (int);  // restituisce la dimensione in byte del tipo specificato

int i=0;
short j=4;
i + j;  // ”j” viene convertito da short ad intero temporaneamente per svolgere l’operazione.
j + j;  // anche qui “j” viene trattato come intero perché è il tipo numerico più efficiente.
j = j + j;
// tutte le volte che uso short, char etc in operazioni, essi vengono promossi ad interi e si produce un intero che eventualmente può essere riconvertito in short se viene assegnato a short.

const int SIZE = 100;
int ai[SIZE];  // viene allocato uno spazio di 100; lo spazio da occupare deve essere definito nel momento della dichiarazione
ai[10] = 34;  // alla funzione si passa il puntatore al primo elemento del vettore (decadimento del tipo)
//avviene questo decadimento per evitare di dover copiare l'intero array nel momento del passaggio per valore)
//anche in ai[10] = 34; decade il tipo a puntatore

ai[j] == *((ai)+(j));  //true; anche la seconda è una giusta scrittura del linguaggio
// * va a leggere nel puntatore calcolato
&ai[0] == &(*(ai)+(0)) == & (* ai) = ai;
```

```
for (int i=0; i < SIZE; ++i)
  fai_qualcosa_di_intelligente_con(a[i]);  // ”a[i]” è considerato in automatico un puntatore; nel caso successivo no quindi devo specificare

for (int* pi=ai; pi != ai + SIZE; ++pi)  //pi non è considerato un puntatore quindi devo specificare
  fai_qualcosa_di_intelligente_con(*pi);

int* p = &SIZE;  // Errore perché io potrei modificare liberamente il puntatore ma SIZE è const quindi:
const int* p = &SIZE; //puntatore varia, il valore resta costante  ->  const (int* p) = &SIZE;
int* const p = &SIZE; //puntatore costante, valore variabile  ->  int* (const p) = &SIZE;
const int* const p = &SIZE;  //a sinistra del * si parla dell'oggetto puntato, a destra si parla del puntatore  ->  oggetto puntato * puntatore  ->  const (int* const (p)) = &SIZE;

const int* q;
{
  const int* p = &SIZE;
  q = &p;
}  //termine del blocco
std::cout << *q;
//Il puntatore “q” punta ad un valore che però al termine del blocco smette di esistere e quindi il puntatore "penzola"
//Il puntatore deve essere cancellato prima del valore puntato.

int* pi = new int(18);  //new = creami una porzione di memoria di intero, inserisci 18, e il puntatore punterà a tale oggetto
delete pi;  //elimina oggetto puntato ma non il puntatore
pi = NULL;  //punta a vuoto
pi = nullptr;  //nuovo standard c++

namespace pippo {
  int a = 7;
}
...
namespace pippo {
... //qui vale ancora la definizione di a
}

int funz(...) {
  using pippo::a;  // la ridefinizione vale fino al termine di questo blocco
  using namespace pippo;  // utilizza tutte le dichiarazioni fatte in pippo
  static int a=0;  // vale anche dopo la chiusura del blocco fino al termine del programma
  ++a;
}
```

FINE esempi

Il carattere è la cosa più piccola indirizzabile in c++.
Il bool varia ma è almeno 1 byte (in alcune macchine è anche 4 byte).
Nella libreria std vengono definite classi chiamate trace come la classe numeric_limit (#include <limits>).
Il codice generico è un codice che funziona indipendentemente dal dato usato e per fare ciò bisogna sapere i limiti di un tipo.
- Se si converte un intero a floating point si potrebbe avere errore di arrotondamento se il numero intero è troppo grande.
- I floating point vanno usati solo in 2 casi:
1) non vi interessa molto il risultato finale
2) state usando un algoritmo in cui si sa che l'errore è contenuto entro un certo range
- Esempio di instabilità dei float: lo 0,1 che è sempre arrotondato.
Bisogna conoscere il motivo per cui certi costrutti esistono per capire se vanno usati o no.

Array

Array è un tipo di dato composto (formato da più dati).

++i --> calcola il valore e poi modifica lo stato del programma.
È meglio il preincremento (per leggibilità perché si capisce prima cosa vuole fare il programmatore).

- Un array si può indicare con il range [ai, ai + size)
- * expr: estrae l'oggetto puntato
- & expr: estrae l'indirizzo di un oggetto
- I puntatori permettono di creare array che non debbano avere la dimensione specificata nel momento della dichiarazione.

Scope di una variabile = DOVE è valida
Cliclo di vita di una variabile = QUANDO nasce e quando muore

Ciclo di vita

- **Allocazione statica** = variabili globali (in certi casi anche locali). Prima di far partire l'esecuzione si alloca la memoria per le variabili globali.
Quando termina il programma, tali variabili verranno deinizializzate (distrutte) e poi deallocate (restituire lo spazio della memoria al sistema).
- **Allocazione automatica tramite stack**: nascono quando ad esse viene data una definizione (fino ad allora sullo stack non ci sono); viene deallocata al termine del blocco.
- **Allocazione dinamica**: gestisco io le risorse del programma; decido io quando far morire un oggetto.
- Io **creo un puntatore** pi ad oggetto, ma alla chiusura di un blocco il puntatore muore e quindi non ho più modo di trovare l'oggetto puntato.
Si risolve questo problema facendo delete pi;. Questo comando elimina l'oggetto puntato ma NON il puntatore.
Memory lik = quando si cancella il puntatore senza cancellare l'oggetto puntato.

Campi d'azione

- namespace (campo d'azione globale).
exstern std::ostream cout; --> variabile dichiarata a livello di namespace (scatole).
- int a=7; --> namespace globale (senza nome).
- Una variabile globale è presente ovunque (se la includo).
Eccezioni: se la a viene ridefinita all'interno di un altro blocco, allora la seconda a nasconderà quella globale fino al termine del blocco.
Variabile statica (static): vale come globale (dopo che è stata dichiarata)

Puntatori e Riferimenti

```
int a;
int* pa = &a;
int& ra = a;
const int& ra = a; //posso fare modifiche su ra ma non su a
*pa; //restituisce l'oggetto puntato
ra; //si riferisce già all'oggetto
```

- Col puntatore si ha a che fare con due oggetti (puntatore e oggetto puntato).
- Col riferimento si ha solo un oggetto (che è l'oggetto a cui punta).
- Il puntatore occupa memoria, il riferimento no (l'oggetto occupa sempre memoria).
- Il riferimento non può riferirsi a nulla, il puntatore si.
- Il riferimento è "appiccicato" allo specifico oggetto, il puntatore invece può puntare ad altri oggetti diversi.

Quando passo i valori ad una funzione come li passo?
- in c SOLO per valore (copia del valore passato)
- in c++ si può passare sia per valore sia per riferimento (la funzione userà un suo nome per riferirsi al valore passato)

```
void fun(int a) //qualsiasi modifica fatta su a non si riperquote all'esterno
void fun(int* pa) //le modifiche fatte sul puntatore non si riperquotono sull'esterno, ma quelle sull'oggetto si
void fun(int &a) //si modifica anche all'esterno l'oggetto a
```

Si usa sempre il passaggio per valore tranne quando:
- voglio che la modifica fatta al valore passato si ripercuota sull'esterno.
- se il valore è troppo grande meglio non copiarlo quindi usare il passaggio per riferimento o puntatore.

Puntatori o riferimento?
- i puntatori sono scomodi
- i riferimenti sono più semplici dato che hanno un solo oggetto
- passare il puntatore da l'impressione che la funzione modifichi il puntatore (bisogna specificare se è nullo o no); con i riferimenti non esistete il caso "nullo.
- uso il puntatore solo se necessario (spostarlo).

Tipi predefiniti scalari e non scalari

Non scalari: array, struct/class; insieme di oggetti dello stesso tipo.
Scalari: valore numerico singolo e le operazioni effettuabili su di esso.

- Deprecato usare ++ su un bool, cioè meglio evitare anche se è possibile farlo senza errori
- Array: blocco di oggetti di dimensione determinata
- struct / class: di default nella struct i valori sono public, nella class invece private.

```
struct Razionale;
struct Razionale{
  int num;
  int den;
};
Razionale r;
int fun() {
  r.num;
}
Razionale* pr = &r;
int bar() {
  *pr.num; //sbagliato
  (*pr).num; //giusto
  pr --> num; //più elegante di quello sopra ma fa la stessa cosa

  a[i] == i[a]; //con i intero ed a array
}

sizeof(r); //è sicuramente maggiore o uguale alla somma dei sizeof dei singoli campi

struct Razionale {
  int num; //4 byte
  char c; //1 byte
  int den; //4 byte
  short p; //2 byte
}
<          4 byte          >  :  num
< 1 byte > < padding 3 byte >  :  c
<          4 byte          >  :  den
< 2 byte > < padding 2 byte >  :  p
```

- Meglio scriverlo ordinando i tipi dal più piccolo al più grande perché nel caso precedente ci sono in totale 5 byte di padding INUTILI:

```
struct Razionale {
  char c;
  short p;
  int num;
  int den;
}
```

< 1 byte > < 2 byte > < padding 1 byte > : c , p
< 4 byte > : num
< 4 byte > : den

- Si ha 1 solo byte di padding (rispetto ai 5 precedenti).

Sulle struct non esiste l'operatore di uguaglianza perché eventuali bit di padding potrebbero essere diversi.

Stringhe

Vettori di caratteri (tipo string fornito da utente).
Enumerato: tipo di dato come insieme di pochi numeri etichettati.

#define

#define: assegna un identificatore (nome) ad un valore.
#undef: toglie un identificatore (nome) ad un valore.
Quando il preprocessore incontra la seguente direttiva:
#define identificatore valore
dove, identificatore è un nome simbolico (che segue le regole generali di specifica di tutti gli altri identificatori) e valore è un'espressione qualsiasi, delimitata a sinistra da blanks o tabs e a destra da blanks, tabs o new-line (i blanks e tabs interni fanno parte di valore).
Sostituisce identificatore con valore in tutto il file (da quel punto in poi).

Es: #define bla frase qualsiasi anche con "virgolette"
Sostituisce (da quel punto in poi) in tutto il file la parola bla con la frase: frase qualsiasi anche con "virgolette" (la sostituzione è assolutamente fedele e cieca, qualunque sia il contenuto dell'espressione che viene sostituita all'identificatore; il compito di "segnalare gli errori" viene lasciato al compilatore!)
In generale la direttiva #define serve per assegnare un nome a una costante (che viene detta "costante predefinita").

Es. #define ID_START 3457
da questo punto in poi, ogni volta che il programma deve usare il numero 3457, si può specificare in sua vece ID_START .
Esistono principalmente due vantaggi nell'uso di #define:
- Se il programmatore decide di cambiare valore a una costante, è sufficiente che lo faccia in un solo punto del programma;
- Molto spesso i nomi sono più significativi e più mnemonici dei numeri (oppure più brevi delle stringhe, se rappresentano costanti stringa) e perciò l'uso delle costanti predefinite permette una maggiore leggibilità del codice e una maggiore efficienza nella programmazione.

#ifdef

#ifdef:
- true: se l'identificatore è stato definito.
- false: se l'identificatore non è stato definito.

#ifndef:
- true: se l'identificatore non è stato definito.
- false: se l'identificatore è stato definito.

Esempio:
#ifndef test //se l'identificatore “test” non è mai stato definito allora...
#define final //definisco “final”
...
#endif //fa terminare la definizione di “final”

Identificatori già usati:

Ogni #include include del codice (anche di classi); un include può includere a suo volta altri include.
C'è il rischio che vengano inclusi due identificatori uguali ma che si riferiscono ad oggetti diversi.
Questa situazione va controllata con #ifndef e #define come si vedrà nell'esempio sottostante.

ESEMPI:

File: Razionale.hh

```
#ifndef guarda_Razionale.hh //controlla che una classe non sia stata inclusa più volte (dando ad ogni file incluso una etichetta)
#define  guarda_Razionale.hh

#include <iostream>
class Razionale {

public:
Razionale(); //default
Razionale(const Razionale& y); //costruttore copia
Razionale& operator=(const Razionale& y); //operatore assegnamento
~Razionale(); //distruttore
//se non li definisco, ci pensa il compilatore
//se scrivo "Razionale(const Razionale& y) = default;" utilizza quello definito dal compilatore (solo in c++ 11)
//"delete;" non utilizza quello di default (solo in c++ 11)

//soluzione A
Razionale(int n);
Razionale(int n, int d);
//fine soluzione A
//se le funzioni fanno la stessa cosa, meglio scrivere così:
Razionale(int n, int d=1); //se non definisco d, la chiamata aggiunge in automatico 1
//se però l'elemento di default è troppo grosso, chiamando la funzione con un solo elemento ogni volta deve creare la d quindi meglio dividere le due funzioni come in precedenza
//Vale anche:
Razionale(int n=0, int d=1); //sostituisce anche il costruttore vuoto "Razionale();"

//supponendo di tenere la soluzione A
explicit Razionale(int n);
//obbliga l'utilizzo del costruttore, impedendo promozioni da parte di altri costruttori
//explicit va usata per tutte le funzioni che si richiamano anche con un solo argomento quindi anche con "Razionale(int n, intd);"
```

```
//metodo 1 op +
Razionale operator+(const Razionale& y) const; //i due const indicano che non vengono modificati i valori che utilizzo

//metodo 2 op +
Razionale operator+(const Razionale& x, const Razionale& y);
//quale usare dei due metodi dipende da come implementerò la classe
//se però il valore l'ho dentro (x), posso accedere ai valori privati della classe e quindi ho più potere
//ma se per l'implementazione so che x non la dovrò usare, allora meglio il 1° metodo

Razionale& operator+=(const Razionale& y); //Razionale& permette concatenazione; void invece non permette la concatenazione; è meglio evitare fare la concatenazione
void operator-=(const Razionale& y);
void operator*=(const Razionale& y);
void operator/=(const Razionale& y);
const Razionale& operator++(); //prefisso;  ++r;
Razionale operator++(int) { //postfisso con un intero che viene completamente ignorato; usato solo per differenziare i due operatori ++;   r++;
//Altro metodo
Razionale& x = *this; //this = puntatore; *this = oggetto puntato
Razionale tmp = x;
++x;
return tmp;
}
//Si avrebbe conflitto perché le due funzioni ++ hanno lo stesso nome, stesso tipo, stesso numero di argomenti; per differenziarle si mette (int) per indicare qual è il postfisso; stessa cosa con il --

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

bool OK() const; //metodo che contiene i controlli che indicano se un oggetto soddisfa le proprietà invarianti

};

std::ostream& //restituisco uno ostream così posso concatenare
operator<<(std::ostream& os, const Razionale& r); //uso & per motivi di efficienza; const perché l'utente non vuole che si modifichi; se non metto const, l'interfaccia pretende che r venga modificato (ed essendo const, da errore). Se mettete const potete stampare sia quelli const che quelli non const.

//Da sistemare la posizione della dichiarazione di funzione
Razionale //usare il riferimento qui, non funziona, meglio usare la copia
operator+ (const Razionale& x, const Razionale& y) { //il += costa meno del + quindi nell'operatore + uso l'operatore +=
  Razionale tmp=x; //costruttore di copia che potevo anche scrivere "Razionale tmp(x);"
  tmp += y;
  return tmp;
}
```

```
//vale anche per -=,*=,/= ma anche per eventuali matrici, vettori etc quindi si possono fare dei copia e incolla

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

bool
operator> (const Razionale& x, const Razionale& y) {
  return x>y;
}
//gli altri operatori di relazione si costruiscono su questi due (vale anche per altri tipi di dato, non solo per razionale)
```

Qui essendo in Razionale.hh NON BISOGNEREBBE FARE LE DEFINIZIONI (come invece abbiamo fatto); bisogna farle in Razionale.cc e poi includere Razionale.hh.

```
#endif // fine "guarda_Razionale.hh"
```

File: Razionale.cc

```
#include "Razionale.hh"
#include <cassert>

Razionale::Razionale() { //fa delle operazioni in più per costruire il num_ e den_ (creo un oggetto che poi butto via)
  num_ = 0;
//qui non posso fare l'assert perché non so ancora cosa sia den_; qui l'invariante non vale
  den_ = 1;
//Fai la cosa giusta (TM).
  assert(OK()); //assert si usa durante il debugging (controlla che il costruttore costruisca cose giuste)
}

Razionale::Razionale() : num_(0), den_(1) { //questa è migliore della precedente perché non fa copie inutili (ricordare che l'ordine che devono avere num e den è quello di come sono stati dichiarati)
  assert(OK());
}

Razionale
Razionale::operator--(int) {
  Razionale& x = *this;
  assert(x.OK()); // chiede che all'esterno venga passato qualcosa che sia effettivamente un razionale
```

```
  Razionale tmp = x;
  assert(tmp.OK()); //controlla se noi (implementatori) abbiamo sbagliato

  --x;
  assert(x.OK());

  return tmp;
}

//controlli in OK
bool
Razionale::check_inv() const {
  #ifndef NDEBUG
  const char* prefix = "l'invariante non soddisfa la classe";
  #endif

  if(den_ <= 0) {
    #ifndef NDEBUG
    std::cerr << prefix << "denominatore minore o uguale a 0";
    #endif
  }
}
```

File: main

```
#include "Razionale.hh"

void foo(Razionale r); //prova
void fu(bool); //prova

bool
test01() {
//costruttore di default dei Razionali restituisce il razionale 0.
  Razionale r1;
  Razionale r2(1234);
  Razionale r3(2,3);
  return false;

  foo(true); //posso convertire il true in un intero questo perché i costruttori con un solo elemento,
effettuano una conversione automatica del valore che gli passo come nel caso qui mostrato (da true
a int)
//per rimediare si modifica il costruttore con explicit che richiede il tipo esatto

  std::cout << r1; //si può fare con la sola dichiarazione dell'operatore << fuori dalla classe anche di
tipo void
  std::cout << r1 << "il mio razionale"; //la dichiarazione dell'operatore deve essere ostream&
```

```
  r1 = r2 = r3; //funziona se l'operatore = è dichiarato Razionale&
//se dichiaro l'operatore = void allora potrò fare SOLO un assegnamento senza concatenazioni (r1=r2;)

  x = ++y + r; //una sola operazione modifica due variabili y e x quindi si rischia l'undefined behaviour; EVITARE

  int i = 5;
  foo(++i, ++i); //implementation undefined: non so quale valore attuale viene letto prima e quindi non so quale delle due i è 6 e quale è 7; potrebbe anche dare undefined behaviour
}
```

FINE esempi

Cosa lasciamo fare all'utente?
Provo tutti i possibili operatori che l'utente potrebbe voler usare

Operatori

- operatori aritmetici binari: -,+,*,/
- operatori aritmetici unari: -r, +r (+r non è un l-value (left value); è il risultato di una operazione quindi r-value)

Se r fosse un short, nel momento in cui lo uso per una operazione (+r) vengono trattati come int

- operatori di assegnamento: +=, -=, *=, /=
- operatori di incremento: ++r, r++, --r, r--
- operatori di confronto (relazionali): ==, !=, <, <=, >, >=

Ci siamo concentrati su const: devo sapere se un valore è modificabile o no.

- Sapere se dobbiamo passare per valore o riferimento.
- Ci sono operazioni in cui si lavora con oggetti già creati e altre che invece creano nuovi oggetti.

- Definiamo la forma normale (base)

Nei razionali:
Per lo 0 si usa la forma 0/1.
Per lo 0 "a basso" restituisco errore.
...

- Sfrutto le proprietà che ho deciso e implementato (proprietà invarianti di classe in determinati punti).
- In certi punti tali proprietà potrebbero essere violate ma in modo transitorio; alla fine (fuori dalla classe) sono sicuramente sempre rispettate.
- Le proprietà invarianti vanno controllate spesso ma se il programma è grosso, si rischia un grande rallentamento (i controlli sono inutili ad esempio quando si hanno elementi const o se in entrata è già stato controllato il valore), se nel codice non ci sono "magheggi" (codici strani) allora si può evitare il controllo (sono inutili anche dopo l'utilizzo di una funzione che ho già dichiarato in precedenza nella quale l'invariante è già stato controllato).
- Ogni controllo (assert OK) fa risparmiare tantissimo tempo sul controllo complessivo (molto utili negli algoritmi con cicli o con ricorsione).

Certe volte però le invarianti non bastano.

Differenza tra inline e macro
Sebbene le funzioni inline siano simili alle macro, le funzioni inline vengono analizzate dal compilatore, mentre le macro vengono espanse dal preprocessore.
Differenze:
- Le funzioni inline seguono tutti i protocolli di indipendenza dai tipi applicati alle funzioni normali.
- Le funzioni inline sono specificate utilizzando la stessa sintassi di qualsiasi altra funzione, con la differenza che includono la parola chiave **inline** nella dichiarazione di funzione.
- Le espressioni passate come argomenti alle funzioni inline sono valutate una sola volta. In alcuni casi, le espressioni passate come argomenti alle macro possono essere valutate più volte.
Inline: è migliore per richiamare più volte nel main la stessa funzione.

One definition rule (ODR)
Tipi: devono essere definiti una volta per ogni unità di traduzione (sintatticamente e semanticamente uguali).
Valori: una sola definizione (modificabile più volte ma è sempre una).
Funzioni: una sola definizione inline.

Codice sintatticamente diverso e semanticamente uguale:

```
//codice 1
typedef int INT
class Razionale {
  INT num_;
  INT den_;
}
//codice 2
class Razionale {
  int num_;
  int den_;
}
```

Problema inclusioni ripetute:
- Pippo definisce la classe Razionale -> Razionale.hh
- Pluto definisce la classe Polinomio (con coefficienti razionali) -> Polinomio.hh (che include Razionale.hh)
- Paperino scrive codice che usa i razionali e include Razionale.hh.
ma usa anche i polinomi includendo Polinomio.hh (che include anche Razionale.hh).
Violazione perché ho dichiarato due classi razionale.

Problemi legati ai nomi:
- Si potrebbero usare lo stesso nome per entità diverse (in scope diversi).
- Overloading: stesso nome per riferirsi ad entità diverse (nello stesso scope).
L'overloading si può avere solo per nome di funzioni; nel caso della classe si può usare namespace per differenziare i nomi.
- L'overloading viene risolto dal compilatore ma noi non sappiamo come e questo è un male.
- Risoluzione overloading (generica): passi che un compilatore copie per sapere quale funzione richiamare.

RISOLUZIONE OVERLOADING (per ogni singola chiamata)

1) **Funzioni candidate**: il compilatore decide quali sono le funzioni candidate alla chiamata.
2) **Funzioni utilizzabili** (viable): alcune candidate potrebbero essere buone o no; esclude quelle non utilizzabili.
3) **Funzione migliore utilizzabile**: si seleziona la migliore (se esiste).

1) **Candidate**: si guarda
- il **nome**.
- **visibilità**: deve essere visibile nel punto in cui noi la chiamiamo.
- Non mi interessano i tipi o il numero dei valori passati per essere candidatata

```
namespace A {
  void foo();
}
void foo();
foo(); //quella interna al namespace non è visibile quindi richiamo quella globale
using A::foo(); //uso quella interna ad A
```

ADL (Argument Dependent Lookup): se nella chiamata negli argomenti esiste un tipo di dato definito dall'utente all'interno del namespace "Pippo" allora in automatico il namespace "Pippo" si apre e mostra tutte le funzioni contenute.
In caso contrario il namespace non si apre.

```
namespace A {
  void foo();
  class R{};
}
namespace B {
  void foo();
}
void foo();
R var; //apre il namespace A
foo(var); //come candidata ho anche il foo dentro ad A
```

- Posso disabilitare ADL ad esempio:

```
B::foo(var); //così la A non si apre
```

- Mi possono fregare quando:

```
namespace A {
  void foo(int);
  namespace B {
    void foo(double);
    void foo(float);
    foo(5.3); //5.3 è un DOUBLE di default
    //questa foo appena trova le due foo float e double si ferma e quindi non vede la foo int; questo perché il namespace applica un hiding (nasconde i nomi delle altre funzioni)
    }  //fine namespace B
}  //fine namespace A
```

- Altro esempio:

```
class B {
  void foo(double);
};
class D : public B {
  void foo(int);
}
D d;
d.foo(5.3); //vede la foo di D e basta
```

2) **Utilizzabili**: si guarda
- **Numero e tipo argomenti**: troppi non va bene, pochi dipende se sono stabiliti dei valori di default
- Si considerano le **conversioni implicite** per capire il numero e tipo degli argomenti.

3) **Migliore**: si guarda
- Ne esiste una sola migliore (vince quella).
- Non ne esiste una migliore (ambigua).

Ambiguità: si fanno n confronti, uno per ogni argomento di una funzioni e vince la funzione che in tutti gli altri confronti non ha mai perso e ha vinto almeno una volta.
Per risolvere le ambiguità si utilizzano le conversioni; di seguito sono indicate in ordine dalla migliore alla peggiore, le:

Conversioni

0) **Nessuna conversione**.
1) **Corrispondenze esatte**
2) **Corrispondenza per promozione**
3) **Conversione standard**
4) **Conversioni definite dall'utente**

1) **Corrispondenze esatte** (anche se non lo sono esattamente).
- conversioni Lvalue -> Rvalue
- conversioni array -> puntatore
- conversioni funzione -> puntatore

Esempi:

```
void f(const char*);  //1
void f(char*);  //2
void f(bool);  //3
const char* cp;
f(cp);  //corrispondenza ad 1
f(0);  //ambigua
f("ciao");  //corrispondenza ad 2
f(true);  //corrispondenza ad 3
f(static_cast<char*>(cp));  //corrispondenza ad 2
```

Lvalue: rappresenta un oggetto che può essere indirizzato.
Rvalue: è un valore che non può essere indirizzato (il valore di un'espressione / una variabile temporanea).

La funzione si aspetta Rvalue come argomenti per i parametri passati per valore (se gli argomenti sono nomi di variabili (Lvalue) devono essere convertiti a Rvalue).
La funzione si aspetta Lvalue come argomenti per i parametri passati per riferimento.
Esempi:

```
#include<string>
string color("rosso");
void print(string);
int main() {
   int a,b,c;
   c=a+b;  //l'operatore + richiede una conversione da Lvalue in Rvalue
   print(color);  //corrispondenza esatta: conversione, da Lvalue in Rvalue
   return 0;
}
```

Conversioni di qualificazione
Riguarda solo i puntatori; si tratta di una conversione che aggiunge il qualificatore const oppure volatile al tipo puntato.
- Const: una variabile nasce con un valore che non può essere modifcato (se lo si modifica si va in undefined beavoir).
- Volatile: dice che una variabile si può modificare senza che il programma lo sappia.
Questo serve per registi hardware (sensore che legge la temperatura cambia continuamente il dato passato) oppure multytrading (più tread che scrivono nella stessa memoria; ogni tread non sa quale modifica ha fatto un altro).
Esempio:

```
int *a, *b;
bool is_equal(const int *, int *);
int main() {
   if(is_equal(a, b)) cout << "true";
   else cout << "false";
}  //stampa true perché effettua una conversione di qualificazione
```

2) **Corrispondenza per promozione**
Se il tipo dell'argomento non corrisponde esattamente al tipo del parametro, allora lo si promuove:

Da	A
boolean, char, unsigned char, short	int
unsigned short	int, unsigned int
float	double

Gli argomenti enumerati (contatori) vengono promossi al più piccolo tipo integrale che può contenerli nell'ordine: int, unsigned int, long, unsigned long.
Esempi:

```
enum colore { bianco : 100, nero : 1000, viola : OxFFFFFFFF};
void f(unsigned int);  //1
void f(int);  //2
void f(char);  //3
f(u'a');  //corrisponde a 2
f(0);  //corrisponde a 2
f(true);  //corrisponde a 2
f(viola);  //corrisponde a 1
```

3) **Conversione standard**
Se il tipo dell'argomento non corrisponde al tipo del parametro, né esattamente né per promozione, allora si applicano le regole di conversone standard:
- Conversioni intere: ogni argomento di tipo integrale o enumerazione è convertito in un altro tipo integrale.
- Conversioni in virgola mobile: ogni argomento di tipo floating è convertito in un altro tipo floating.
- Conversione virgola mobile-intero: ogni tipo integrale è convertito ad un tipo floating e viceversa.
- Conversione di puntatori: 0 è convertito in un tipo puntatore; ogni tipo puntatore è convertito a void.
- Conversioni booleane: Ogni argomento di tipo intero, tipo in virgola mobile, enumeratore o puntatore è convertito al tipo bool.
Esempi:

```
void f(unsigned int);  //1
void f(float);  //2
void f(void*);  //3
char *cp;
f('a');  //ambigua, corrisponde a 1 e 2
f(0);  //ambigua, corrisponde a 1, 2, e 3
f(3.14);  //ambigua, corrisponde a 1 e 2
f(cp);  //corrisponde a 3
```

4) **Conversioni definite dall'utente**
- Gli operator.
- Da razionale ad altro razionale.

Regole dei template (overloading):
- I template non sono funzioni ma schemi di funzioni da cui ricavo la funzione da usare.
La funzione che ricavo da un template è diversa dal template e quindi la devo scrivere.
- Se da un template ottengo (istanzio) una funzione candidata, essa è anche già utilizzabile.
- Sfinae: non riuscire a fare la sostituzione nei parametri del template non è un errore.
- In caso di ambiguità, si escludono le funzioni ottenute tramite istanziazione (quelle templatiche).
In pratica si hanno come candidate una funzione templatica e una non templatica, si considera quella non templatica.
- Avendo due funzioni templatiche come in questo caso, si sceglie quella più specializzata cioè quella il cui insieme di tutte le istanze che una funzione può generare, è un sottoinsieme delle istanze che può generare l`altra funzione.
Esempio:

```
template <typename T>
void f(T t1, T t2);  //funzione 1
template <typename T, typename U>
void f(T t, U u);  //funzione 2
```

La 1 è più specializzata perché:
- l'insieme delle possibili istanziazioni della 2 comprende sia l'insieme in cui T=U (che è l'insieme 1) più tutte le istanziazioni in cui T è diverso da U.
- L'insieme della 2 contiene la 1 quindi la 1 è più specializzata.

ESEMPIO 1)
- Non ci sono namespace quindi è più semplice

```
void f(const char* s);   //funzione 1
template <typename T>
void f(T t);  //funzione 2
template <typename T>
void f(T t1, T t2);  //funzione 3
template <typename T, typename U>
void f(T t, U u);  //funzione 4
template <typename T>
void f(T* pt, T t);  //funzione 5
template <typename T, typename U>
void f(T* pt, U u);  //funzione 6

template <typename T>
void g(T t, double d);  //funzione 7
template <typename T>
void g(T t1, T t2);  //funzione 8
int test() {

  f('a');
//Candidate: tutte le funzioni che si chiamano "f" (1-6) anche se i template non sono funzioni ma schemi di funzione quindi posso utilizzare quello schema per ottenere una funzione candidata.
//Utilizzabili: quali funzioni hanno un parametro carattere o convertibile in carattere?
2 (void f<char>(char);  → funzione ottenuta dallo schema templatico)
Non utilizzabile: 1 (non posso convertire un char in un *char), 3 (chiede due parametri), 4,5 e 6 per il motivo della 3.
//Vince la 2:  void f(const char* s);

  f("aaa");
//Candidate: da 1 a 6, 2 (istanza: void f<const char*>(const char*) )
//Utilizzabili: 1, 2 passa l'array per valore e si ha un decadimento di tipo;
1 e 2 sono identiche e sembrerebbe ambigua, ma secondo le regole tra una funzione templatica e una non templatica, vince la non templatica.
//Vince la 1:  void f<const char*>(const char*)

int i;

  f(i);
//Candidate: 1 candidata (ma non utilizzabile), 2 istanziata ( f<int>(int) ) ed è l'unica utilizzabile; questa 2 è diversa dalla istanza del caso precedente.
//Vince la 2:  f<int>(int)

  f(i, i);
//Candidate: scarto la 2 e 1 (perché hanno un solo argomento); candido la 3,4,5,6
//Utilizzabili: 3 deve avere due tipi uguali, la 4 invece lascia la possibilità di avere tipi diversi (senza obblighi); 3 e 4 creano entrambe un'istanziazione e quindi generano ambiguità MA anche qui si
```

```
utilizza una regola dei template che fa vincere la funzione più specializzata.
- 3 è più specializzata della 4 e 5
- 5 è più specializzata della 4 e 6
La 5 e la 6 però vogliono un puntatore quindi falliscono.
//Vince la 3:  void f<int>(int,int);

  f(i, &i);
//3 fallisce la deduzione; 4 genera qualcosa; 5 e 6 non vanno bene.
//Vnce la 4:  void f<int, int*> (int, int*);

  f(&i, i);
//3,5 falliscono; con la 4 e 5 si può fare la specializzazione (candidate)
//La 5 è più specializzata della 4
//Vince la 5:  void f<int>(int*,int);

  double d;

  f(i, d);
//La deduzione di 3 fallisce (non si può fare la conversione standard da double a int nel secondo
parametro).
//Fallisce anche con la 5 e 6.
//La 4 riesce e la istanzia.
//Vince la 4:  void f<int, double>(int,double);

  f(&d, i);
//Candidate: 6 e 4.
//La 6 è più specializzata della 4.
//Vince la 6:  void f<double, int>(double*,int);

//D'ora in poi le candidate saranno solo la 7 e 8
  long l;

  g(l, i);
//La 7 e la 8 non sono confrontabili; 8 non si può usare (servono due argomenti di tipo diverso);
posso istanziare la 7 con la conversione implicita standard da int a double;
//Vince la 7:  void g<long>(long,double);

  g(l, l) ;
//vince la 3: void g<long>(long, double) della 7
//void f<int, double>(int, double) istanza 8
  g(l, d) ;

  g(d, d) ;
//ottengo due istanziazioni 7 e 8 e ho ambiguità quindi errore; non so quale invocare (
}
//non introduco il concetto di specializzazione nelle funzioni perché utilizzo l'overloading
servono regole supplementari per gestire le funzioni template.
FINE esempio 1)
```

ESEMPIO 2):

```
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); //chiama la 2 (perché nella 1^ deve fare una conversione che nella 2 non c'è)
  }
} // 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) { // funzione #8
    f(d); // cand:5 No Argument dependent look up; c'è una trasformazione da r-value a l-value
    g(c); // cand:3,6 Applico Argument dependent look up per aprire il namespace Na e perché ho
due funzioni g --> non ce né una migliore quindi ambiguità
    h(c); // cand:4,7 vince la 4
  }
} // namespace NB
void f(NA::C c, NB::D d); //9
void test3(NA::C c, NB::D d); //10
f(1.0); //cand: non c'è funzione utilizzabile (c'è una sola candidata la 9)
g(); // cand: non c'è funzione utilizzabile (e nessuna candidata)
g(c); //cand: si apre NA e si trova un unica candidata che è anche la migliore
g(c, d); //cand: si apre la NA e NB, si trovano due candidate ma c'è ambiguità
```

FINE esempio 2)

FINE Overloading

```
b.print(std::cout); //ho due membri 'b' e 'std::cout' quindi due argomenti da controllare
void print(std::ostream&) const; //conversione di qualificazione perché ho messo il const
void print(std::ostream&); //dopo aver risolto l'overloading si fa il controllo di accesso con il quale
si trova un errore perché ho accesso a parti private
```

- se ho due funzioni una private e una public, ma vince la private, mi da errore anche se esiste la public.
- le funzioni della classe che non hanno static hanno un argomento in più cioè l'oggetto della classe.

```
const Base* pb2 = static_const<const Base*>(pd);  //conversione
```

CORREGGI ERRORI

Codice assegnato:

```
template <typename T>
class Set : public std::list<T> {
public:
// Costruisce l'insieme vuoto.
Set();
Set(T t);
Set(Set y);
void operator=(Set y);
virtual ~Set();
unsigned int size();
bool is_empty();
bool contains(Set y);
T& min();
void pop_min();
void insert(T z);
void union_assign(Set y);
void intersection_assign(Set y);
void swap(Set y);
std::ostream operator<<(std::ostream os);
private:
// ...
};
```

Codice corretto:

```
template <typename T>
class Set : private std::list<T> { //corretto uso private al posto di public
public:
// Costruisce l'insieme vuoto.
Set();
explicit Set(const T& t); //corretto, explicit per essere sicuro che non venga trattato come conversione; aggiunta di const e &
Set(const Set& y); //corretto Set(Set y); loop infinito che richiama se stesso
void operator=(const Set& y); //corretto
virtual ~Set(); //non conosciamo virtual
unsigned int size() const; //corretto con const riferito all'oggetto implicito this
bool is_empty() const; //corretto
bool contains(const Set& y) const; //corretto
T min() const; //corretto; ho tolto il riferimento perché con esso do libero accesso a delle parti essenziali che sono fondamentali per la mia invariante
const T& min() const; //corretto rispetto a quello sopra per evitare la copia
void pop_min();
void insert(const T& z); //corretto
void union_assign(const Set& y); //+= corretto
void intersection_assign(const Set& y); //corretto
void swap(Set& y); //corretto (scambia)
```

```
private:
// ...
}; //fine classe
//std::ostream operator<<(std::ostream os);
std::ostream& operator<<(std::ostream& os, const Set<T>& y) const; //corretto
```
FINE correzione

EXCEPTION SAFE (codice scritto bene)

Esempio eccezioni:

```
String EvaluateSalaryAndReturnName( Employee e ) //invoco il costruttore di copia (forse eccezione)
{
  if( e.Title() == "CEO" || e.Salary() > 100000 ) //2 chiamate, 2 possibili eccezioni; se non c'è più memoria lancio 1 eccezione; 1 eccezione su ==; altre 3 eccezioni per la parte dopo il ||
  {
    cout << e.First() << " " << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

Cammini di esecuzione:
- 3 normali: quelli dell'if quando 1 V, 1 V e 1 F, o 2 F.
- + molte altre eccezioni che generano altri percorsi.

Esempio - da correggere

```
void {
  File* fa = apri_file("A");
  File* fb = apri_file("B");
  fai_qualcosa(fa,fb);
  chiudi_file(fa);
  File* fc = apri_file("C");
  fai_qualcosa(fb,fc);
  chiudi_file(fb);
  chiudi_file(fc);
}
```

- Se trovo un errore (eccezione) in apri_file io programmatore non devo fare niente (è compito di chi lo utilizzerà gestire l'errore).
- Se la prima apri_file va a buon fine, tu ottieni una risorsa ed ora è sotto la tua responsabilità.
- Se la seconda invocazione di apri_file, non va a buon fine, la funzione termina ma senza eseguire la chiudi_file di A (errore); stessa cosa con eccezioni su invocazioni successive.
- Anche chiamando la chiudi_file, se da eccezione, si ha errore ma in tal caso non so quale altra funzione potrei richiamare; questo problema capita con funzioni che devono liberare le risorse, le quali NON devono avere eccezioni, altrimenti non so appunto come restituire le risorse al sistema.

Distinzioni tra operazioni:
- Di acquisizione risorse.
- Che lavorano su risorse.
- Che rilasciano risorse (non devono generare errori).

Correzione 1)

Usiamo try-catch

1) Se il try contiene tutto, il compilatore può dare errore perché nel catch potrebbe voler eliminare la fa o fb o fc anche se non sono ancora state prese come risorsa.

2) Il try va messo dopo l'acquisizione di fa e il catch alla fine (tramite try-catch annidati si applica ciò a tutte le acquisizioni).
3) Il punto 2) non basta perché catturando una eccezione annidata essa non si ripercuote sull'esterno quindi bisogna rilanciarla per fare in modo che sia catturata dai blocchi esterni.
Uso il comando throw; che generalmente vuole una espressione ma se si vuole rilanciare la stessa identica eccezione che è già stata catturata, si può scrivere senza espressione (solo throw).

Con queste modifiche si rende il codice neutrale rispetto alle eccezioni.
Il throw si mette anche nell'ultimo per indicare l'errore anche a chi utilizza la funzione.

Correzione che gestisce ogni acquisizione con un try-catch (blocchi annidati)

```
void {
  File* fa = apri_file("A");
  try{  //1
    File* fb = apri_file("B");
    try{  //2
      fai_qualcosa(fa,fb);
      chiudi_file(fa);
      File* fc = apri_file("C");
      try{  //3
        fai_qualcosa(fb,fc);
        chiudi_file(fb);
        chiudi_file(fc);
      } catch(...) {
          chiudi_file(fc);
          throw;
      }  //fine try 3
    } catch(...) {
        chiudi_file(fb);
        throw;
    }  //fine try 2
  } catch (...) {
      chiudi_file(fa);
      throw;
  } //fine try 1
} //fine void
```

- Cattiva gestione risorse.
- Si comporta bene (o in modo sensato, senza rubare risorse al sistema) in assenza o presenza di eccezioni.
- Lo scopo era rilasciare le risorse all'uscita dal blocco ma così il codice è poco leggibile quindi bisogna usare i distruttori di una classe.
- In java si creano degli oggetti (puntatori); una volta che si smette di utilizzarli, essi smettono di puntare ma rimangono; poi di tanto in tanto passa la garbage collection che elimina le risorse non utilizzate (lo fa lui senza che tu possa sapere quando e come lo fa).

Correzione 2)

Exception safe: scrivere il codice bene significa anche togliere il maggior numero di try-catch.
- Per creare una variabile si usa il costruttore.
- Per rilasciare la risorsa si usa il distruttore (implicitamente senza nemmeno scrivere codice).

Il distruttore è il candidato ideale a risolvere i problemi di rilascio risorse quindi possono essere utilizzati al posto dei try-catch creando però una classe specifica a riguardo che gestisca le risorse.

Idioma RRID: le risorse rilasciate vengono distrutte.
RRID = Resource Release Is Destruction.
Idioma RAII: l'acquisizione delle risorse deve essere effettuata come una inizializzazione.
RAII = Resource Acquisition Is Initialization

Correzione che utilizza classe RAII

```
class File_RAII {
public:
  File_RAII(const char* nome) //si può evitare il riferimento perché costa poco
    : my_file(apri_file(nome)) {
  }

  File* get() const{
    return my_file;
  }

  ~File_RAII() {
    chiudi_file(my_file);
  }
private:
  File* my_file;
}

void {
  File_RAII fa("A");
  File_RAII fb("B");
  fai_qualcosa(fa.get(),fb.get());
  File_RAII fc("C");
  fai_qualcosa(fb.get(),fc.get());
}
```

- Niente try-catch e codice anche più breve (abbiamo applicato l'exception safe).
- In java non posso lasciar fare la garbage collection quindi devo eliminarli io con try catch.
- In java esiste il codice try-finally che viene eseguito sempre e che viene usato come un try-catch. unico che cattura tutto.
- La classe RAII è una classe di appoggio per evitare try-catch.

Correzione 3)

```
class File_RAII { //classe gestore delle risorse
public:
  explicit File_RAII(const char* nome) //in certi casi si potrebbe anche evitare l'explicit
    : my_file(apri_file(nome)) {
  }

  operator File*() const{ //conversione implicita
    return my_file;
  }

  ~File_RAII() {
    chiudi_file(my_file);
  }
private:
  File* my_file;
  File_RAII(const File_RAII&); //Dichiarazione e non implementato (non puoi usare costruttore di copia)
  File_RAII& operator=(const File_RAII&); //Dichiarazione e non implementato;
}
```

- Questa classe ha 3 funzioni esplicite ; implicite ci sono quelle di copia e assegnamento quindi altre 2 che dobbiamo ridefinire per evitare possibili errori.
- Dato che questa classe deve gestire dati esterni ad essa, bisogna ridefinire tutte le funzioni implicite tra cui: copia, assegnamento, costruttore, distruttore oppure posso dire all'utente di non usare mai il costruttore di copia e assegnamento con una dichiarazione senza implementazione.

```
void {
  File_RAII fa("A");
  File_RAII fa2(fa); //usa il costruttore di copia implicito che genera due puntatori ad A che pensano di essere gli unici; inoltre invocando il distruttore di oggetto 2 volte, si cerca la seconda volta di distruggere un oggetto che non esiste → errore causato dalla mancanza del costruttore di copia
  File_RAII fb("B");
  fb = fa; //stessi problemi di File_RAII fa2(fa); con la differenza che in questo caso rimane aperto un file sparso per il sistema → errore peggiore del precedente causato dalla mancanza del costruttore di assegnamento
  fai_qualcosa(fa.get(),fb.get());
  File_RAII fc("C");
  fai_qualcosa(fb.get(),fc.get());
}
```

- Nei puntatori spesso si usano classi (RAII) per "vestirli".
- Le classi RAII sono anche templatiche ma di solito sono specifiche.
- Il codice ottenuto ora però non è uguale a quello precedente perché in questo il rilascio avviene tutto alla fine mentre nell'originale (iniziale) il file fa viene rilasciato prima del termine della funzione.
- Il metodo con RAII utilizza una politica LIFO, ma il programma originale no (rilascio la fa prima di fb).

- Per risolvere il problema "LIFO" bisogna chiedere all'utente se posso modificare il suo codice di partenza scrivendolo il politica LIFO così:

```
void {
  File* fb = apri_file("B"); //fb
  File* fa = apri_file("A"); //fa
  fai_qualcosa(fa,fb);
  chiudi_file(fa); //fa → X
  File* fc = apri_file("C"); //fc
  fai_qualcosa(fb,fc);
  chiudi_file(fc); //fc → X
  chiudi_file(fb); //fb → X
}
```

- Oppure posso creare un sotto blocco per chiudere fa PRIMA della chiusura finale, in questo modo:

Codice corretto (finale):

```
class File_RAII {
public:
  explicit File_RAII(const char* nome)
  : my_file(apri_file(nome)) {
  }

  operator File*() const{
    return my_file;
  }

  ~File_RAII() {
    chiudi_file(my_file);
  }
private:
  File* my_file;
  File_RAII(const File_RAII&); //costr. copia, dichiarato ma non inizializzato
  File_RAII& operator=(const File_RAII&); //costr. assegnamento, dichiarato ma non inizializzato
}

void {
  File_RAII fb("B");
  {
    File_RAII fa("A");
    fai_qualcosa(fa.get(),fb.get());
  }
  File_RAII fc("C");
  fai_qualcosa(fb.get(),fc.get());
}
```

STACK

Livelli di correttezza diversi per soddisfare livelli di sicurezza diversi.

Livelli di exception safe:

- Livello basso: se posso sempre distruggere un oggetto (che genera errore ma che potrebbe essere modificato prima dell'eliminazione).
- Livello strong (forte): o va tutto a buon fine o niente.
- Livello massimo: non fallisce mai.

Livello strong (con stack)

```
T& top();
const T& top() const;  // entrambi in caso gli venga passata uno stack const o modificabile
bool OK() const;  // (rileva errori)
new T[capacity == 0 ? 16 : capacity]; //if; costruisce 16 elementi di tipo T;
// L'operatore new fa due operazioni 1) prende la memoria vuota 2) parte e costruisce gli oggetti uno alla volta fino al 16º
inline
Stack::Stack(const size_type capacity) //costruttore
: vec_(new T[capacity == 0 ? 16 : capacity]),
capacity_(capacity == 0 ? 16 : capacity),
size(0)
{
  assert(OK());
}
//P.S. vec, capacity, size sono i campi interni alla classe Stack che posso definire subito dopo il costruttore della classe, tramite l'utilizzo dei :
```

Possibili errori di questo codice:

- Se la new non riesce a trovare abbastanza memoria vuota, da errore senza prendere le risorse quindi noi non abbiamo problemi.
- Se la new trova i terreni vuoti e incomincia a costruire ma poi trova un errore nell'ennesimo oggetto, essa torna indietro e ci pensa lei ad assicurare l'exception safe.
- Se l'errore è generato dal distruttore di T la colpa è di chi ha creato T.
- La capacity non trova errore.
- Size nemmeno.

Livello strong soddisfatto.

- Il distruttore deve rilasciare le risorse che sono sotto il diretto controllo del programmatore.
- Swap scambia il contenuto dei due stack; è sufficiente scambiare i 3 dati membri (utilizzo la swap della std); ciò non genera eccezioni perché si fanno solo assegnazioni non copie.
- pop diminuisce la size dopo aver controllato (ma resta compito del utente passare uno stack non vuoto).

```
inline Stack:Stack(const Stack& y) : vec_(make_copy(y.vec_ , (y.size_ , y.size == 0 ) ? 1: y.size)) , capacity_((y.size.....
```

La "**make copy**" si prende la risorsa formata dagli elementi che devo copiare e la capacità dell'elemento che devo copiare.

- Se la makecopy è exceptionsafe allora lo è anche il costruttore di copia.
- Dopo il controllo dell'assert, allochiamo la risorsa e creiamo un puntatore a oggetti di tipo T con la

capacity copiata.
- Gli oggetti che andiamo a copiare sono complicati a piacere, quindi potrebbero generare eccezioni (gestite con try-catch).
- Se per caso parte un'eccezione dobbiamo distruggere quello che abbiamo creato lanciando la delete sul puntatore che abbiamo creato.
(la delete: prima passa a distruggere la memoria, riportandola a un deserto di bit restituendola al sistema).
Rilanciamo l'eccezione trovata per esser exception safe.

Operatore di assegnamento: se la copia e lo swap erano exception safe allora creiamo un temp che salva il contenuto di y e poi si fa una swap tra le risorse e il temporaneo (restituisco la dereferenziazzione di this).
Il temp verrà distrutto alla chiusura del blocco.

La **push()** a volte deve fare un ulteriore allocazione di risorse.
vec_[size] = emel potrebbe generare un eccezione ma non dobbiamo preoccuparci noi; lo devono prevedere quelli che hanno fatto la classe T

Documentazione (Doxygen)
//! --> documenterà una singola linea successiva.
/*! --> documentazione più dettagliata anche del blocco (marcatori, commenti speciali).
Doxygen --> genera in forma quasi automatica la documentazione del software.
La documentazione a parte (separata dal codice), causa un disaccoppiamento tra codice e spiegazione (se modifico il codice potrei poi dimenticarmi di modificare la documentazione).
Doxygen è usato maggiormente per le interfacce.
Doxugen in pochi comandi produce qualcosa di professionale e utile.

TEMPLATE

```
inline int
max(int a, int b) {
  return a < b ? b : a;
}
//funziona solo per gli interi.

#define min(a,b) ( (a) < (b) ? (b) : (a) )
// lavora con tutto (anche se è più rischiosa).
```

- Risolviamo questo problema con la parametrizzazione tramite template

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

- Questo è una template di funzione (questa è una macchina che genera il codice a seconda del tipo che gli passo).

```
int main() {
  max<int> (4,6); //istanza
  max<double>(4,6); //istanza che genera codice diverso dal precedente
}
```

Type dependent: parte di codice che dipende dal tipo di un parametro
Poi ci sono altre porzioni di codice (in template) che invece possono essere controllate subito

L'errore type dependent si troverà solo durante la compilazione (durante l'istanziazione).
Le funzioni templatiche possono essere definite più volte.

- Gli errori quando si usano i template, si possono verificare molto lontano rispetto alla posizione nel codice sorgente.
- Lo schema della funzione max non funziona ad esempio se uso il tipo string o dei puntatori a char (const char*)
- Se in una classe non metto public, private etc allora un oggetto razionale alloca prima il denominatore e poi il numeratore e quindi usare i puntatori non sarebbe assurdo ma comunque da non fare.

- In definitiva, se per certi tipi il template non funziona allora dovrò specializzare il template

specializzazione di template di funzione:

```
template <>
inline const char*
max<const char*> (const char* a, const char* b) {
return strcmp(...);
}
//questo codice gestisce il tipo const char*
//posso solo se ho prima dichiarato la funzione max con template
```

- così però è lungo da scrivere se poi devo farlo anche per molti altri tipi

- Se nel main scrivo:
max(4,6); //in automatico riconosce che max dovrà essere di tipo int
max(4,3.2); //non funziona perché ho due valori di tipo diverso quindi max non sa quale tipo diventare
max("pippo","pluto"); //riconosce max di tipo const char* e funziona (ma è un caso particolare)
int i = max(4.3,6.5); //max di tipo double; risolve e restituisce un double che poi sarà convertito in int

- bisogna sempre dire come istanzio una classe (sempre max<double>(...))
(nel c++ 2003 tutti i parametri devono essere di tipo T - forse sbagliato)
- ora posso specializzare e specificare alcuni parametri

VECTOR

```
#include<vector>
int main() {
  std::vector<int> vi; //sta istanziando la classe vector col tipo int; inizializzo con il costruttore di base
  std::vector<int> vj(100, 5); //vettore di 100 elementi con valore 5; se non metto 5, alloca solo 100 elementi (tutto ciò con un solo costruttore explicit con parametri di default)
  std::vector<int> vk = vi; //costruttore di copia specifico che da per scontato che vk sia dello stesso tipo di vi
// la && usata con un tipo (vector&&) indica un r-value
  std::vector<double> vd (vj.begin() + 20, vj.begin() + 30); //possibile grazie al costruttore templatico specificato in seguito
}
```

- il template vector ha due parametri di cui il secondo è definito di default (allocatore).
- l'iteratore è un elemento astratto che implementa la scansione tra i vari elementi del vector (vai avanti di 5 posizioni, prendi l'elemento in posizione p, etc)

```
//nell'include vector
template <typename _InputIterator>
vector (_InputIterator __first, _InputIterator __last, const allocator_type& __a = allocataor_type())
: _Base(__a)
//costruttore templatico (posso crere una copia a partire da qualunque iterator; invece di usare tipi diversi uso gli stessi indipendentemente dal tipo degli elem del vector)
```

- Si possono avere classi non templatiche con all'interno funzioni templatiche.
- Ci sono puntatori che possono stare anche sul limite (fine) ma in tal caso non possono ne leggere ne scrivere ma possono tornare indietro per vedere l'ultimo elemento
- auto: assegna automaticamente un tipo ma nel caso in cui si possa richiamare un costruttore const o non const, richiama il non const; in tal caso bisogna differenziare le funzioni con una 'c' così potrò scegliere quale effettivamente richiamo (begin(), cbegin()).
- end() in un vettore di 1 elemento punta al 2° elemento.
- push_back: aggiunge in fondo il valore passato.
- pop_back: butta via l'ultimo elemento se esiste e senza restituire quello che ha cancellato per exception safe.

- Il vector ha un contratto molto exception safe esclusi costruttori di molti elementi.
- Rimuovere un elemento alla fine costa molto meno che toglierlo all'inizio (togliendolo all'inizio un valore, tutti gli altri li devo spostare indietro di uno).
- Con il metodo intert si può inserire nel vector un elemento in una determinata posizione.
- C'è anche la insert che inserisce un valore più volte (n volte).
- C'è anche la insert templatica che permette di inserire in una certa posizione una certa quantità di elementi.
- Stessa cosa per la cancellazione con l'erase.
- swap: scambio tra vector.
- clear: toglie tutto lasciando però la memoria allocata e vuota.

CODA A DOPPIA ENTRATA

- size_type: quantità solo positiva.
- difference_type: differenza tra due posizioni che può essere anche negativa.

Non c'è la capacità perché non ho un blocco fisso di memoria (e non è nemmeno contiguo come invece avviene nel vector).

- pop_front(): toglie l'elemento dalla cima.
- operatore ==: devono avere lo stesso tipo, stessa implementazione come sequenza (se ho uno stack implementato con vector o con coda a doppia entrata, le cose cambiano).

Le operazioni sui vettori sono semplici oltre al fatto che posso accedere alla memoria senza problemi e c'è anche il fatto che essendo tutti collegati, probabilmente un elemento successivo è già in cache.

LISTA

Ricordare costruttore con iteratore templatico.
- due begin (const e non)
- due rbegin (const e non)
- maxsize
- resize

L'operatore + non viene dato; vengono date funzioni per svolgere il +.
Nelle liste possono usare solo l'operatore -- (decremento) o il ++ (incremento).
- Remove_if(_Predicate): il predicato è una funzione che applicata, restitusce vero o falso (se vero, elimina).
- Unique(): se l'elemento dopo è uguale, lo si elimina (si applica in genere dopo un riordinamento).

Predicato binario: riceve due oggetti.
Perché abbiamo la unique nelle liste? Perché sono operazioni frequenti ed è una unique generica già implementate bene.
Ma se non ho bisogno dell'unique allora la dovrei mettere fuori (quindi l'unique sporca un po' l'interfaccia essendo in certi casi inutile).

Tre contenitori sequenziali **vector, lista, coda a doppia entrata** con interfacce simili ma non uguali.

Se usate degli iteratori avrete del codice che funziona su qualunque sequenza perché non gli interessa come è fatta, gli interessa solo che sia una sequenza (cioè che abbia certe regole base).

Iteratori (simili ai puntatori):
1) Migliori (dei puntatori): puntano effettivamente a qualcosa e posso fare qualsiasi cosa con essi.
2) Non deferenziabili ma fatti bene (posso decrementarli come end(), confrontarli, etc).
3) Iteratori denglin (penzolanti): puntano a vuoto (l'unica operazione fattibile su di essi è assegnargli un altro oggetto).

Range (coppia di due operatori):
- Se uno dei due iteratori non è valido nemmeno il range lo è.
- I due iteratori devono avere lo stesso tipo, devono puntare dentro lo stesso oggetto, devono puntare in posizioni buone.
- Iteratori e range devono essere validi.
[First, Last) -> range (coppia di iteratori)
- const su iteretor non funziona perché l'iteretor deve essere modificabile.

Esempi situazioni particolari:

```
//prendiamo la push_back che inserisce un elemento in coda, e vediamo dove causa problemi
std::vector<int> vi;
...
vi.push_back(24); //aggiunta
...
std::vector<int>::iterator b=vi.begin();
...
vi.push_back(30); //aggiunta: se ho un vector di 1 sola posizione, avendo già aggiunto il 24, rialloca la memoria in modo tale da poter aggiungere il 30; MA IL b NON PUNTA alla nuovo array riallocato, il b penzola
//bisogna quindi ricalcolare b
...
assert(!vi.empty());
*b = 18; //sbagliato
*(vi.begin())=18; //aggiunta: cosi ricalcolo b
//le operazioni che noi facciamo sui range e iteratori possono renderli non validi
```

Container: nella programmazione ad oggetti, un oggetto contenitore (o semplicemente container) è una classe di oggetti che è preposta al contenimento di altri oggetti. Questi oggetti usualmente possono essere di qualsiasi classe, e possono anche essere a loro volta dei contenitori.
Esempi di classi contenitori: insiemi, liste, stack, code e mappe.

```
template <typename Container, typename Value>
unsigned long
count(const Container& c, const Value& v);

//Definizione 1) della funzione count
template <typename Container, typename Value>
typename Container::size_type
count(const Container& c, const Value& v) {
   unsigned long n=0;
   for(Container::const_iterator i=c.begin(), i_end=c.end() ; i!=i_end ; ++i) {
      if(*i == v)
```

```
            ++n;
    }
    return n;
}

//Definizione 2) della funzione count (migliore)
template <typename Container, typename Value>
typename Container::size_type
count(const Container& c, const Value& v) {
    typename Container::size_type n=0;
    for(typename Container::const_iterator i=c.begin(), i_end=c.end(); i!=i_end ; ++i) {
        if(*i == v)
            ++n;
    }
    return n;
}

#include<vector>
int main() {
    std::vector<int> vi(150,17);
    std::vector<int>::size_type num17 = count(vi,17); //Non si usa typename, non è una scatola nera;
Se lo scrivi quando non ce bisogno potrebbe dare errore in base al compilatore
    std::cout <<num17<<std::endl;;
}
```

Altri esempi (iteratori):

```
iterator
insert (iterator pos, const value_type& x) { //restituisce un iteratore cioè quello nuovo, ricalcolato
dopo l'inserimento all'inizio di un nuovo elemento
}

//tutore (chiede a quello sotto), specializzazione totale
template <typename _Iterator>
struct iterator_trits
{
  typedef typename _Iterator::iterator_category iterator_category;
  typedef typename _Iterator::value_type value_type;
  ...
};

//specializzazione parziale di template di classe (siamo noi a rispondere)
//specializzazione parziale fattibile solo con i template di classe
template <typename _Tp>
struct iterator_traits<_Tp*>
{
  typedef _Tp value_type;
  typedef ptrdiff_t difference_type;
...  };
```

```
template<typename Iter, typename value> //Funzione – Conta elementi
(unsigned long) //serve più generico
(typename Iter::difference_type) //non compila
count(Iter first, Iter last, const value& v) {
  (unsigned long n=0;)  //meglio il successivo
  typename std::iterator_trits<Iter>::difference_type n=0;
  for( ; first!=last; ++first)
    if(*first == v)
      ++n;

  return n;
}

template<typename Iter, typename value> //Funzione – Trova elemento
Iter
find(Iter first, Iter last, const value& v)
{
  for( ; first!=last; ++first)
    if(*first == v)
      retunr first;

  return last;
}

#include <vector>
#include <iostream>
main() {
  std::vector<int> vi(150,17);
  int ai[]={1,2,3};
  std::vector<int>::size_type num17 = count(vi,17);
  std::vector<int>::size_type num17_bis = count(vi.begin(), vi.end(), 17);
  std::vector<int>::size_type num17_ter = count(ai, ai+3, 17);
}
```

- una classe che implementa un iteratore deve contenere al suo interno dei typedef
- numeri climits (tutori): danno informazioni sui tipi base. Dobbiamo creare una cosa simile per la classe templatica.
- tutore = iterator_traits

Per fare un codice templatico bisogna sapere qual'è il tipo usato per una certa variabile.

```
template <typename Iter1, typename Iter2> //Funzione – Controlla uguaglianza
bool
equal (Iter1 first1, Iter1 last1, Iter2 first2) { //last2 non lo passo perché do per scontato che sia maggiore o uguale a last1 quindi non mi serve
  for( ; first1 != last1 ; ++first1, ++first2) {
    if(!(*first1 == *first2)) { //diverso oppure not uguale? meglio il == perché certe volte in un tipo non viene definito in un operatore != invece == è un operatore necessario in un tipo che utilizza confronti
```

```
      return false;
    }
  ++first2;
  }
}
```

- Togliere gli spazi di fine riga è meglio (sono completamente inutili)
- Gli iteratori hanno vari tipi (iteratori su array, liste etc)
Mi viene passato un iteratore e voglio fargli fare un passo di 150 in avanti, come faccio in modo generico?
- Lo faccio per l'array (e vale solo per quello)
- Lo faccio per le liste che vada avanti un passo alla volta (applicabile a tutto ma nel caso dell'array si avrebbe spreco).
Posso chiedere ad un tipo cosa è in grado di fare per poi fare la cosa migliore in automatico?
- Negli iterator def esisteva la iterator category (categoria dell'iteratore).
Interroghiamo la categoria e poi scegliamo quale codice eseguire.

Categorie degli iteratori

- struct **input_iterator_tag** { }
Iteratore con cui si può solo leggere in una sequenza (operazioni: creare iteratore, copiare, assegnare, distruggerlo, confrontarlo con == e != (no > <), incrementarlo di 1 (no decremento), leggere il valore (no scrivere).
- struct **forward_iterator_tag** : public input_iterator_tag {}
So fare tutto quello che fa l'input iterator in più può scrivere (col permesso; se il contenitore è passato come costante, allora non può scrivere ma c'è differenza con l'input perché il forward è stabile perché posso rileggere gli stessi elementi quanto voglio; l'input invece fa l'input e consuma ciò che usa quindi non c'è garanzia che l'elemento precedente sia ancora lo stesso)
- struct **bidirection_iterator_tag** : public forward_iterator_tag {}
Può anche spostarsi indietro di 1 (decremento).
- struct **random_iterator_tag** : public bidirection_iterator_tag {}
Può decrementare e incrementare più di 1 alla volta e fare confronti < > (e i restanti).

Lista: bidirection.
Array: random.
_tag: è solo un tag, poi deciderai tu a cosa serve; essi però possono avere ereditarietà.

Vogliamo creare una funzione templatica che prende un iter e lo fa avanzare di n posizioni (lista o array).

Funzioni templatica (con iteratore)

```
template <typename Iter>
Iter
advance_aux(Iter start, unsigned n, random_access_iterator_tag) {
  return start + n;
}
```

```
template <typename Iter>
Iter
advance_aux(Iter start, unsigned n, input_iterator_tag) {
while (n>0) {
  ++start;
  --n;
}
return;
}

template <typename Iter>
Iter
advance(Iter start, unsigned n) {
  typename std::iterator_traits<Iter>::iterator_category la_mia_categoriy;
  return advance_aux(start, n, la_mia_category);
}
```

- Servirebbero almeno 5 versioni; noi ne abbiamo fatte solo 2 con una 3^ che è quella che verrà richiamata e che sceglierà in automatico quale tra le altre 2 è la migliore da usare (in automatico, in base al tipo del parametro tag).
- Ne servirebbe una 3^ ausiliaria per la versione della lista che va sia avanti che indietro.

Esempio (input operator che non è forward)

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

int main() {
  std::istream_iterator<int> i(std::cin);
  std::istream_iterator<int> i_end;
  std::vector<int> vi(i,i_end); //input

  for(int i=0; i<vi.size(); i++)
    std::cout<<vi[i]<<std::endl;

  std::ostream_iterator<int> out(std::cout); //com'è esattamente
  std::ostream_iterator<int> out(std::cout,"a capo \n"); //ogni output aggiunge un pezzo "a capo \n"
  std::copy(vi.begin(),vi.end(), out); //output

  return 0;
}
```

Piccoli difetti di questi metodi:
- dato un iteratore non si può arrivare al contenitore quindi non si può eliminare da un contenitore degli elementi se di quel contenitore conosco solo l'iterator.
- Bisogna capire bene come rendere ancora più generici gli algoritmi fatti fin'ora.

Ora li abbiamo parametrizzati in base ai valori e alla sequenza; vedremo come parametrizzarli in base alla funzione (comportamento) che hanno.

Passo una funzione come argomento di una funzione

```
template <typename _InputIterator, typename _Predicate>
typename iterator_traits (_InputIterator __first, _InputIterator __last, _Predicate __pred)
{
  typename iterator_traits<_InputIterator>::difference_type __n=0;
  for( ; __first != __last; ++first)
    ...
}
```

- Gli passo un predicato.
- I puntatori a funzioni sono dei predicati (sono callable).
- Esistono anche gli **oggetti funzione**: classi o struct che hanno un nome (tipo).

```
bool pari(int n) { //funzione considerata
  return (n%2) == 0;
}

struct Pari {
   bool operator() (int n) const { //overloading della chiamata funzionale; doppia coppia di parentesi
     num_calls++;
     return (n%2) == 0;
   }

  bool operator() (const Razionale& r) const { //oggetto funzione
    num_calls++;
    return (n%2) == 0;
  }

  static unsigned num_calls = 0;
}; //fine struct

int main() {
  std::vector<int> voti;
  //popola vettore
  unsigned long num_pari = std::count_if(vi.begin(), vi.end(), Pari); //conosce solo l'indirizzo di una funzione (pari) che prende in input un intero
  unsigned long num_dispari = std::count_if(vi.begin(), vi.end(), dispari); //la pari istanzia la funzione, la dispari usa la funzione già generata

  unsigned long num_pari2 = std::count_if(vi.begin(), vi.end(), Pari()); //sa che qui Pari è un oggetto (schema di funzione) che può istanziare una funzione in base a ciò che gli passi dentro
  unsigned long num_disparipari2 = std::count_if(vi.begin(), vi.end(), Dispari());
```

- Il Pari istanzia una funzione diversa dalle precedenti; nel caso Dispari devo generare un altra versione diversa dalla Pari con ottimizzazioni diverse e specifiche a seconda di chi la chiama quindi:
- con oggetto funzione, il compilatore può ottimizzare la funzione (dato che ne esistono 2)
- con oggetto funzione, si ha una maggior occupazione di spazio

- (difetto) è fastidioso andare a vedere quale funzione (all'interno della funzione oggetto) sto usando; andavano create delle collable.

```
}  //fine main
```

Oggetto funzione: oggetto il cui tipo è una classe qualsiasi l'importante è che al suo interno ci sia operator().
- L'oggetto funzione a differenza di una funzione può contenere al suo interno uno stato (quante volte è stato chiamato num_calls, etc).
- Istanziazioni con ottimizzazioni diverse e specifiche.

Lambda expression (funzioni collable)
lamba x . x (prende x restituisce x)
lamba x . x+1 (prende x restituisce x+1)
- Nel c++ 2011 hanno dato una sintassi per queste lamba expression:

```
[](int x) { return x; } //espressione di una funzione lambda che restituisce x
[](int x) { return x+1; }
[](int x) { return (n%2) == 0; }
```

Per conoscere il tipo in c++ 2011 si può usare "auto" (auto i = 2;)

```
template <typename T>
return? sum(T a, T a);
```

- Si risolve con:

```
template <typename U, typename T>
U sum(T a, T a);
```

- Però dovrei scrivere

```
sum<int>(4,5);
```

- Allora faccio

```
template <typename T>
decltype(a+b) sum(T a, T b);
```

- non può essere eseguita perché non si conosce il tipo di a e b

```
template <typename T>
auto sum(T a, T b); -> decltype(a+b);
```

- Dichiara DOPO il tipo della funzione

```
auto sum(int a, int b) {
  return a+b;
}
```

- Il compilatore riconosce in automatico il tipo
- I linguaggi cambiano nel tempo è questo ne è un esempio

```
auto la_mia_lamba = [](int n) { return (n%2) == 0; } //per poterla riutilizzare; “auto” è necessario
//passerò "la_mia_lamba" come valore di una funzione
```

Esercizi - FORNIRE IMPLEMENTAZIONE

Fornire l'implementazione di un algoritmo (funzione) che sia applicabile ad un contesto il più generico possibile.
Sotto quale condizioni l'utente può invocare quel algoritmo?
(vedi algoritmi stl come esempio).

Esercizio 1):
- Prende in input una sequenza e una funzione da applicare a tutti gli elementi (quindi due parametri (first e last) e la funzione).

```
template <typename _InputIterator, typename _Funcition> //deve essere InputIterator e collable
_Function  //restituisco "_Funzione" perché esso potrebbe contenere qualcosa nel suo stato interno (num_calls)
for_each(_InputIterator  __first, _InputIterator  __last, _Function  __f)
{
  for( ; __first != __last; ++__first)
    __f(*__first);  //funzione che si applica agli elementi
  return (__f);
}
```

Esercizio 2):
- Date due sequenze (di cui la seconda è più lunga o uguale alla prima), confrontare gli elementi.
- Esempio dell'stl:

```
template <class _InputIter1, class _InputIter2>
pair<_InputIter1, _InputIter2>
mismatch(_InputIter1 __first1,    _InputIter1 __last1,    _InputIter2 __first2) {
  …
  while (__first1 != __last1 && *__first1 == *__first2) {
    ++__first1;
    ++__first2;
  }
  return pair<_InputIter1, _InputIter2>(__first1, __first2);
}
//Dovrebbe essere una funzione ricorsiva dato che chiama se stessa.
```

CAST

Il cast è una conversione esplicita
- stile C:
(type) expr;

```
int I = 3.67; //potrei non averla fatta di proposito la conversione da double a int
int I = (int) 3.67; //l'ho fatta di proposito
```

- Stile C++;
Stessa cosa del C
- NON USARE MAI tranne in casi idiomatici

```
(void) foo(a,b,c);  //non interessa il risultato e quindi dico che voglio un void (non ha tipo); evita di avere errori del tipo "errore, non so di che tipo è il risultato".
int i = static_cast<int>(3.67);   //questo va bene
//static_cast: converte un oggetto expression nel tipo di type-id, esclusivamente sulla base dei tipi presenti nell'espressione.
static_cast <type-id> ( expression )

void foo(const vector<int>& vi) {
  std::vector<int>& v = const_cast<std::vector<int>&> (vi);   //const_cast  toglie il const
  v[10]=45;
}
//const_cast è molto pericoloso, meglio non usarlo; lo si usa in casi particolari ad esempio con i razionali, si può dover semplificare il valore ma anche se fisicamente cambia, in valore resta uguale.

Cast funzionale:
bool b = bool(56); //introdotto perché si aveva codice templatico e in esso si vuole determinare un certo tipo; poco visibile ma utilizzata praticamente solo quando si scrive codice templatico

//reinterpret cast (il più brutto)
double d = 3.1415;
int* pi = reinterpret_cast<int*>(&d);  //vai a leggere il double esattamente come intero senza semplifiche, e ciò restituisce quindi numeri con poco senso.

B* pb;  //classe base
D* pd = dynamic_cast<D*>(pb);   //classe derivata; convertiamo una classe base in una derivata; è dinamico perché viene fatto durante l'esecuzione
```

Classi templatiche

Esempio 1): out

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

template <typename In, typename Out>
Out
copy(In first, In last, Out out) {
  for( ; first != last; ++first) {
    *out = *first;
    ++out;
  }
  return out; //restituisce uno stream di output così posso concatenare
}

int main() {
std::istream_iterator<int> i(std::cin);
std::istream_iterator<int> i_end;

std:;vector<int> vi(i,i_end);
std::list<int> li;

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

*out = 12345;
++cout;

std::copy(vi.begin(), vi.end(), out);

std::copy(vi.begin(), vi.end(),
                                    std::copy(vi.begin(), vi.end(), out); //Concatenazione

std::copy(vi.begin(), vi.end(), li.end()); //Inserisce gli elementi nella posizione dopo l'ultimo elemento della lista, ma end() è in fondo e su di esso non posso scrivere quindi errore di undefined behavior
//Devo creare io lo spazio prima

li.resize(...);
std::copy(vi.begin(), vi.end(), li.end()); //Non va bene perché normalmente io non so quanti elementi l'utente vuole inserire e quindi non so di quanto fare la resize.
```

- Potrei fare il resize automatico il quale però rischia di occupare molto spazio inutile.
- Gli operatori di output sono overwrite (quando aggiungo qualcosa si cancella quello che c'era prima) ma io voglio aggiungere in fondo (senza cancellare i dati precedenti).
- Dato un iteratore, posso trasformarlo in un iteratore che funziona come inserimento? Dipende dove voglio inserire.

- Se ho a disposizione un contenitore (lista, array etc), posso usare lo stream di output per inserire in fondo dei dati? Si
Esistono classi templatiche chiamate inseritori (insert_Iterator):
- back_insert_Iterator: inserisce in fondo
- front_insert_Iterator: inserisce all'inizio
- insert_Iterator inserisce dove vuoi

std::copy(vi.begin(), vi.end(), std::back_insert_Iterator<std::list<int> >(li)); //Molto lungo e noioso da scrivere; esiste infatti una funzioncina (back_inserter) che ci permette di risparmiare la scrittura di qualche parola.

std::copy(vi.begin(), vi.end(), std::back_inserter(li)); //La back_inserter è usata spesso come aiuto.
//Esiste anche il front_inserter per i front_inserte_iterator
//Dato che i vector non hanno il push_back o push_front, quello che abbiamo detto funziona con le liste e simili.

std::copy(vi.begin(), vi.end(), std::inserter(li,++li.begin())); //inserter inserisce in una parte centrale indicata da "++li.begin()" cioè nella 2^ posizione della lista a partire dall'inizio.
return 0;
}

Overwrite: (es: operatori di output) sta già puntando a qualcosa di valido che poi viene sovrapposto quando voglio scrivere (come quando premo ins sulla tastiera).

Esempio 1): back_insert_iterator

```
template <class _Container>
class back_insert_iterator {
protected:
  _Container* container;
public:
  typedef _Container          container_type;
  typedef output_iterator_tag iterator_category;
  typedef void                value_type; //se sono void significa che non è importante conoscere cosa fanno e/o restituiscono
  typedef void                difference_type;
  typedef void                pointer;
  typedef void                reference;

  explicit back_insert_iterator(_Container& __x) : container(&__x) {}
  back_insert_iterator<_Container>&
  operator=(const typename _Container::value_type& __value) {
    container->push_back(__value);
    return *this;
  }
  back_insert_iterator<_Container>& operator*() { return *this; }
  back_insert_iterator<_Container>& operator++() { return *this; }
  back_insert_iterator<_Container>& operator++(int) { return *this; }
}
```

Esempio 2): Pari

```
template <typename T1, typename T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;

  pair() :
  first(), second() { //posso usare "pair" SE first e second hanno un costruttore di default
  }

  pair(const T1& a, const T2& b) :
    first(a), second(b) { //Posso usare "pair" SE first e second hanno un costruttore con un parametro
  }

  template <class U1, class U2> //Metodo templatico in una classe templatica; class = typename
  pair /*<T1, T2>*/ (const pair<U1, U2>& p)
  : first(p,first), second(p,second) { //costruisce copie di ad es interi a partire da double
  }
```

- Questo ultimo metodo, può generare anche il costruttore di copia della classe?
No, ed è una eccezione alle regole; se voglio costruire una copia di interi a partire da un altra copia di interi, il compilatore chiamerà il costruttore di copia di default e non questo.
}

```
template <typename T1, typename T2>
inline pair<T1, T2>
make.pair(T1 x, T2 y) {
  return pair<T1,T2> (x,y);
} //Permette di scrivere →

foo(std::make.pair(vi,ld));
```

Ricapitoliamo:
- Un template di classe è uno schema che genera una classe che può contenere:
dati, dichiarazioni di tipo, sottoclassi, funzioni, altri schemi templatici che generano altro codice.
- Vengono generati tutti insieme? NO; l'istanziazione delle classi avviene on the main (durante, a seconda di quello che serve); inizialmente però si istanzia tutto quello che sicuramente servirà.
- Le funzioni vengono generate solo se ne abbiamo bisogno.
- Se in una classe ho tante funzioni, ma ne serve solo una, allora generò solo quell'una; questo permette al compilatore di generare meno codice.
- Se una funzione (ad es) richiede che debbano esserci i costruttori di default ma io non voglio usare quella funzione, allora il compilatore non mi darà errore.
Per questo è più faticoso trovare gli errori (bisogna provare tutte le funzioni, fare molti test).
Ma, di positivo, posso usare una classe che, se anche necessita di 50 requisiti, io gli passo oggetti di un tipo che usa solo le funzioni che richiedono solo 25 di requisiti.

Classe templatica (stack)

- Prendiamo la classe stack non templatica e la modifichiamo rendendola templatica (così è più semplice evitare errori).
- Se però abbiamo uno stack di interi non possiamo convertirlo in double.

Per fare ciò bisogna cambiare qualcosa:

- Il costruttore di copia non si può fare templatico (perché altrimenti la classe userebbe quello di default) quindi si usa un altro costruttore che prende uno stack di tipo U e lo trasforma in tipo T.

```
template <typename T>
template <typename U>  //doppio template, caratteristica particolare
inline
Stack<T>::Stack(const Stack<U>& y) :   // Problema 1): stack di T può accedere agli elementi di U solo se è amico (friend)
  vec_(make_copy(y.vec_, y.size_, (y.size_ == 0 ? 1 : y.size_)), // Problema 2): make_copy non compila perché prende un puntatore a T e restituisce un puntatore a T e quindi va templatizzata in modo che prenda un puntatore a U e restituisca un puntatore a T
  capacity_((y.size_ == 0) ? 1 : y.size_)
  size_(y.size_) {

  assert(OK());
}
```

Risolviamo il Problema 2):

```
template <typename U> //Ora la makecopy è utilizzabile nella precedente funzione
static T* make_copy(const U* src, size_type src_size, size_type dst_capacity) {
  assert (dst_capacity >= src_size);
  T* dst = new T[dst_capacity];
  try { //proviamo a copiare gli elementi da src a dst; se si può bene altrimenti errore (in automatico)
    for(size_type i=0; i<src_size; i++)
    dst[i] = src[i]
  }
  catch {
    delete[] dst;
  }&
}
```

Risolviamo il Problema 1):

Esiste un modo per dire che la Stack di T è amica di tutte le altre istanze di Stack (U compresa).

```
template <typename T> friend class Stack;
```

Ricapitoliamo:

Abbiamo ottenuto una programmazione con polimorfismo cioè codice che assume diverse forme.

Polimorfismo statico dei template:

1) L'istanziazione avviene durante la compilazione con aspetti positivi e negativi.
- Positivo: il compilatore può applicare ottimizzazioni molto spinte.
- Negativo: tutte le istanze devono essere note al momento di compilazione; poca libertà all'utente (e codice anche molto lungo).

2) Puoi scrivere codice generico che si può adattare alle situazioni più generiche possibili.

- Questo permette di chiedere implicitamente poche caratteristiche (esempio, se dentro la funzione c'è src[i] significa che il tipo di src implicitamente deve poter aver definito l'operatore [] e SOLO quello, il resto non conta).
- È meglio usare == perché si trova più spesso in un tipo, quindi è più probabile che funzioni.

Esistono anche polimorfismi dinamici, ma di solito sono meglio i template.

Questi accorgimenti però non bastano e quindi servono concetti astratti per costruire classi che si adattino ancora meglio:
Iteratori: concetto astratto di qualcosa che sa fare certe operazioni (esempio, "iteratore sequenza" deve solo essere una successione di elementi).
Queste entità astratte (iteratori) sono anche stati formalizzati un po' meglio:

[vedi stl]

```
template <typename _ForwardIterator1 __first1, typename _ForwardIterator1 __last1,
                  _ForwardIterator2 __first2, typename _ForwardIterator2 __last2)
{
  ... _ForwardIteratorConcept <_ForwardIterator1> ... //Concetto che deve essere soddisfatto
  ... std::__iterator_category(__first)...  //Funzione ausiliaria che prende un iteratore e costruisce l'oggetto di una certa categoria e serve anche per stabilire quale funzione va usata in automatico.
}
```

Problematiche
- gestire le risorse (senza errori)
- dare alla risorsa una interfaccia diversa (da Stack a Coda)

Esempio (Stack): casi particolari di creazione, distruzione, allocazione, deallocazione

```
Stack_Impl::~Stack_Impl() {   // Distruggiamo gli elementi effettivamente presenti su vec_ , in ordine inverso rispetto a quello di costruzione.
  for (size_type i = size_; i-- > 0; )
    vec_[i].~T();  // Distrugge l'elemento SENZA deallocare la memoria
  operator delete(vec_);  // Applica SOLO la deallocazione della memoria
}

Stack::Stack(const Stack& y) : impl(y.impl.capacity_) { // Effettua SOLO l'allocazione di memoria grezza necessaria
  for (size_type i = 0; i < y.impl.size_; ++i) {
    new (impl.vec_ + i) T(y.impl.vec_[i]); // Questa new particolare inizializzarla la memoria con le copie degli elementi contenuti in y, SENZA allocarla
    ++impl.size_;  // E` importante che size_ venga aggiornato passo dopo passo perché in ogni momento deve dire con precisioni quanti sono gli elementi effettivamente costruiti dentro a vec_ così in caso di errore, so esattamente quanta memoria riassegnare al sistema.
  }  //fine for
}

void Stack::push(const T& elem) {
  if (impl.size_ < impl.capacity_) { // Se c'è spazio a sufficienza: faccio new di piazzamento (particolare) e incremento di size_
    new (impl.vec_ + impl.size_) T(elem);
    ++impl.size_;
  }
  else {  // Altrimenti occorre riallocare lo Stack: creiamo un temporaneo e poi facciamo lo scambio. In questo modo, se qualcosa va storto, il temporaneo viene distrutto correttamente e lo Stack iniziale non viene toccato.
    assert(impl.size_ == impl.capacity_);
    Stack temp(impl.capacity_ * 2 + 1);
    for (size_type i = 0; i < impl.size_; ++i)
      temp.push(impl.vec_[i]);

    temp.push(elem);
    swap(temp);
  } // Il temp si elimina alla chiusura del blocco else

}
```

Fine Esempio

Esempio 1):

Ho uno Stack di vector; ogni elemento è di tipo 'oggetto pesante'; cosa succede se dico che voglio uno steck di 500 stack di 'oggetti pesanti'.

1^ versione Stack: costruivamo tutte e 500 steck di 'oggetti pesanti'

2^ versione Stack: costruiamo uno steck impl che occupa 3 oggetti vec_ , size_ , capacity, con il quale alloco ma non costruisco (più veloce e meno dispendioso).

Esempio 2):
Se utilizzo per poco 2 miliardi di interi ma poi ne ho spesso solo 100, il vector e lo steck versione 1 non offre modo per ridimensionarsi (in negativo, cioè ridurre la propria dimensione; lo Stack di 2^ versione invece può).
- Usare .clear(): cancella i dati ma la capacity resta 2 milardi.

Soluzione riallocamento diminuente (in 2^ versione Stack):
Simile al metodo utilizzato nella precedente "Stack::push" con piccole differenze:
- la capacity_ al posto di aumentare viene diminuita
- la memoria tolta, deve essere SOLO deallocata (senza distruzione dato che su di essa non sono presenti elementi).

Finora abbiamo fatto i contenitori sequenziali della stl.
Esistono anche contenitori associativi: consentono di associare un'informazione ad un'altra (anche quelli sequenziali fanno questa cosa, ma in modo forzato).
Per fare ciò serve una mappa che porta da un dominio ad un altro dominio (mappa che fornisce a seconda dell'indirizzo che gli passo, il nome della persona).
Deve essere efficiente: lo si può fare con una lista ma il tempo per fare ricerche è troppo elevato.

MAPPE

Esercizio - conteggio delle parole di un file

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

int main() {
  //leggere tutte le parole del file mettendole in una mappa.
  std::istream_iterator<std::string>   i(std::cin);
  std::istream_iterator<std::string>   i_end;

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

  str::copy(i, i_end, out);

  return 0;
}
```

Include <map>

Class map della stl

```
template <class _Key, class _Tp, class _Compare, class _Alloc>
class map {
public:
…
  typedef _Key                    key_type;
  typedef _Tp                     data_type;
  typedef _Tp                     mapped_type;
  typedef pair<const _Key, _Tp>   value_type;
  typedef _Compare                key_compare;
…
public:
  typedef typename _Rep_type::pointer pointer;
  typedef typename _Rep_type::const_pointer const_pointer;
  typedef typename _Rep_type::reference reference;
  typedef typename _Rep_type::const_reference const_reference;
  typedef typename _Rep_type::iterator iterator;
  typedef typename _Rep_type::const_iterator const_iterator;
  typedef typename _Rep_type::reverse_iterator reverse_iterator;
  typedef typename _Rep_type::const_reverse_iterator const_reverse_iterator;
  typedef typename _Rep_type::size_type size_type;
```

```
typedef typename _Rep_type::difference_type difference_type;
typedef typename _Rep_type::allocator_type allocator_type;
…
key_compare key_comp() const { return _M_t.key_comp(); }
value_compare value_comp() const { return value_compare(_M_t.key_comp()); }
allocator_type get_allocator() const { return _M_t.get_allocator(); }

iterator begin() { return _M_t.begin(); }
const_iterator begin() const { return _M_t.begin(); }
iterator end() { return _M_t.end(); }
const_iterator end() const { return _M_t.end(); }
reverse_iterator rbegin() { return _M_t.rbegin(); }
const_reverse_iterator rbegin() const { return _M_t.rbegin(); }
reverse_iterator rend() { return _M_t.rend(); }
const_reverse_iterator rend() const { return _M_t.rend(); }

bool empty() const { return _M_t.empty(); }
size_type size() const { return _M_t.size(); }
size_type max_size() const { return _M_t.max_size(); }
_Tp& operator[](const key_type& __k) {
  …
}
void swap(map<_Key,_Tp,_Compare,_Alloc>& __x) { _M_t.swap(__x._M_t); }

pair<iterator,bool> insert(const value_type& __x)
  { return _M_t.insert_unique(__x); }
iterator insert(iterator position, const value_type& __x)
  { return _M_t.insert_unique(position, __x); }
…
void erase(iterator __position) { _M_t.erase(__position); }
size_type erase(const key_type& __x) { return _M_t.erase(__x); }
void erase(iterator __first, iterator __last) { _M_t.erase(__first, __last); }
void clear() { _M_t.clear(); }

…
iterator find(const key_type& __x) { return _M_t.find(__x); }
const_iterator find(const key_type& __x) const { return _M_t.find(__x); }
size_type count(const key_type& __x) const {
  return _M_t.find(__x) == _M_t.end() ? 0 : 1;
}
iterator lower_bound(const key_type& __x) {return _M_t.lower_bound(__x); }
const_iterator lower_bound(const key_type& __x) const {
  return _M_t.lower_bound(__x);
}
iterator upper_bound(const key_type& __x) {return _M_t.upper_bound(__x); }
const_iterator upper_bound(const key_type& __x) const {
  return _M_t.upper_bound(__x);
}
```

```
  pair<iterator,iterator> equal_range(const key_type& __x) {
    return _M_t.equal_range(__x);
  }
  pair<const_iterator,const_iterator> equal_range(const key_type& __x) const {
    return _M_t.equal_range(__x);
  }
  …
};
```

Osservazioni sulla class map

- Istanziamo la mappa passandogli una chiave (Key), e un tipo (Tp) di parametro su cui lavorare; Compare è un callable (puntatore a funzione, oggetto a funzione, lambda expression) che costruisce (ordina) in base al tipo di dato e alla chiave ed ha anche un valore di default; Alloc è l'allocatore
- In pair la key è const perché la key non va toccata altrimenti non è garantita l'invariante (l'ordine).
- Parte public: ci sono tanti typedef, soliti metodi begin e end con versione const e non const, etc (simili alla stl).

Differenze tra vector e mappe:

- Per effettuare l'accesso agli elementi, nei vector esistevano due versioni (const e non const) nelle mappe invece esiste una sola versione const alla quale posso passare tra parentesi quadre una stringa al posto di un valore numerico, la mappa in automatico la cerca e quando la trova restituisce il valore a cui è associata.

In questo caso, non ci sono eccezioni perché se non trova la risorsa la crea.

Esempio: cerca la copia che ha Pippo come primo componente; se non la trova, crea la copia che ha Pippo come primo componente e come secondo elemento crea qualcosa con il costruttore di default.

- mapped type: è il metodo simile al precedente che invece lancia eccezione quando non trova la coppia di elementi.
- insert: inserisce nella mia mappa un value type e restituisce una copia.

Se però il valore esiste già non lo modifica e restituisce quello che c'era già.

Posso anche dargli un riferimento per indicargli esattamente dove inserire il valore senza effettuare ricerche (se però il riferimento non è corretto allora utilizza il metodo spiegato in precedenza con anche la ricerca).

- find: trova un elemento con una specifica chiave.
- lower_bound: restituisce un iteratore al primo elemento maggiore o uguale della chiave data.
- upper_bound: restituisce un iteratore al primo elemento maggiore della chiave data.
- equal_range: restituisce una gamma di elementi corrispondenti alla chiave data.

Usiamo la include <map>

Esercizio - conteggio delle parole di un file

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

```
//Funzione carica – 1
template<typename Iter>
void
carica(Iter first, Iter last, Word_Freq& wf) {
  for( ; first != last; ++first) {
    Word_Freq::iterator pos = wf.find(*first);
    if(pos == wf.end() {
      wf.insert(std::make_pair(*first, 1));  //se non la trovo
    }
    else {
      ++(pos -> second);  //incrementa il valore mappato
    }
  }
}

//Funzione carica – 1 (ridotta)
template<typename Iter>
void
carica2(Iter first, Iter last, Word_Freq& wf) {
  for( ; first != last; ++first)
      ++wf[*first];  //non trova la copia quindi crea in automatico un valore a 0 e ci somma 1
}

//Funzione carica _fw – 2
template<typename Iter>
void
carica_fw(Iter first, Iter last, Freq_Word& fw) {
  for( ; first != last; ++first) {
    Freq_Word::iterator pos = fw.find(*first);
    if(pos == fw.end() {
      fw.insert(std::make_pair(*first, 1));  //se non la trovo
    }
    else {
      ++(pos -> second);  //incrementa il valore mappato
    }
  }
}

//Funzione carica_fw – 2 (ridotta)
template<typename Iter>
void
carica_fw2(Iter first, Iter last, Freq_Word& fw) {
  for( ; first != last; ++first)
      ++fw[*first];  //non trova la copia quindi crea in automatico un valore a 0 e ci somma 1
}
```

```
//Funzione oggetto
struct Lunghezza {
  bool operator() (const std::string& x, const std::string& y) const {
    return x.size() < y.size();
  }
}

int main() {
  //Leggere tutte le parole del file mettendole in una mappa.
  std::istream_iterator<std::string>   i(std::cin);
  std::istream_iterator<std::string>   i_end;

  typedef std::map<std::string, unsigned> Word_Freq;
  typedef std::map<std::string, unsigned, Lunghezza> Word_Freq; //altro caso in cui gli passo un metodo di ordinamento (Lunghezza) che modifica il risultato mostrando le stringhe del file (o testo) in input in ordine di occorrenza e poi di lunghezza.
  typedef std::multimap<unsigned, std::string> Freq_Word;
  typedef std::multimap<unsigned, std::string, std::greater<unsigned>  Freq_Word;  //permette il riordinamento di int al contrario (dal più grande al più piccolo)

  Word_Freq wf;
  carica(i, i_end, wf);

  Freq_Word fw;
  carica_fw(i, i_end, fw);

  for( Word_Freq::const_iterator i = wf.begin(), i_end = wf.end(); i != i_end; i++) {  //stampa in ordine alfabetico
     std::cout << "La parola " << i->first << " occorre numero " << i->second << " volte. \a";
  }

  for( Freq_Word::const_iterator i = fw.begin(), i_end = fw.end(); i != i_end; i++) {  //stampa dal nome che si presenta di meno a quello che si presenta di più
     std::cout << "La parola " << i->first << " occorre numero " << i->second << " volte. \a";
  }

  for( Freq_Word::const_reverse_iterator i = fw.rbegin(), i_end = fw.rend(); i != i_end; i++) {
//stampa dal nome che si presenta di più a quello che si presenta di meno
     std::cout << "La parola " << i->first << " occorre numero " << i->second << " volte. \a";
}

  return 0;
}  //fine main
```

- Mappe e multimappe: non sono gli unici contenitori associativi, esistono anche gli insiemi e multiinsiemi.
- Insieme di stringhe o multiinsieme di stringhe: gli elementi entrano tante volte (ma leggendo, potrei non averli in sequenza).
- Gli insiemi sono ordinati; sono mappe con solo il valore (senza chiave).
- I multiinsiemi sono mappe che hanno solo le chiavi.

Contenitori e algoritmi generici

```
template <typename T>  // Parametrizzazione template per tipo
void foo();

template <typename T, int n>  // Parametrizzazione template per valore
void foo();

template <typename T, int n>
void foo(T (&a)[n]);  // Invoca solo funzioni con array lunghi n

template <typename T, unsigned long n>
inline unsigned long
size(T (&a)[n]) {
  return n;
}  // Restituisce il numero degli elementi; nelle funzioni precedenti passavo il puntatore all'array e quindi dovevo contare gli elementi, qui invece so subito la lunghezza (più veloce).
```

Da qui in poi, funziona nel C++ 2011:

```
template <typename Containers> // Parametrizzazione per contenitore (che contiene gli elementi della sequenza)
void foo(const Container& c) {
  …  // fai qualcosa
  for(auto i = c.begin(), i_end = c.end(); i!=i_end; ++i) {  // Stampa
    std::cout << *i;
  }
}
// “foo” funziona solo per gli oggetti che hanno i metodi begin e end; non funziona con array dato che gli array non hanno questi metodi.
// Sono state aggiunte due nuove funzioni (begin e end) nella stl che permettendo di fare “std::begin(c)” risolvendo la gestione dell’array

// Codice generico
template <typename Containers>
void foo(const Container& c) {
  for(auto i = std::begin(c), i_end = std::end(c); i!=i_end; ++i)
    std::cout << *i;
}
```

```
template <typename Containers> //Funzione precedente più corta ma che fa la stessa cosa.
void foo(const Container& c) {
  for(const auto& elem : c)  //Grazie ad auto, si riconosce in automatico il tipo di elem (c).
    std::cout << elem;
}

template <typename Cont>
auto begin(Cout& c) -> decletype(c.begin()) {  //riconosce in automatico che deve restituire un
oggetto di tipo c.begin().
  return c.begin();
}

template <typename Cont>
auto end(Cont& c) -> decletype(c.end()) {
  return c.end();
}
// Fine codice generico

// Per fare in modo che anche gli array funzionino nel "Codice generico", vanno aggiunte due
funzioni esterne che gestiscono appunto l'utilizzo dell'array.
template <typename T, unsigned long n>
T* begin(T (&a)[n]) {
  return a;
}

template <typename T, unsigned long n>
T* end(T (&a)[n]) {
  return a + n;
}

// Altro
template <template <typename> Cont, typename T> // Non si usa spesso
void foo() {
  Cont<T> ct;
  Cont<char> c_char;
}
```

- I template permettono di eseguire calcoli durante la compilazione, questo renderà molto più veloce il programma compilato.
- Per fare le cose a tempo di compilazione devo però fornire costanti.
- Proviamo un'applicazione di ciò al Fattoriale

Esercizio - Fattoriale

```
template <unsigned long n>
struct Fatt {
  enum Value {
    value = n * Fatt<n-1>::value; //se non dai un valore, esso in automatico da 0 poi alla successiva 1 etc; “value” è costante (in automatico)
  };
};

template<>  // Caso base della ricorsione precedente
struct Fatt<0> {
  enum Value {
    value = 1;
  };
}
```

- Con i template si possono avere ricorsioni.
- Con i template si possono fare anche le condizioni con la specializzazione di template (non così "(n==0) 1 ? ..." ma con l’aggiunta di un caso base templatico).
- Tutto avviene durante la compilazione quindi nel programma finale avrò una costante e NON un calcolo da fare.

```
int main() {
  std::cout << Fatt<10>::value << std::endl;  // ”Fatt<10>::value” è già un valore
  return 0;
}
```

Con i template si fa anche:
- **Introspezione**: si può interrogare un tipo per sapere quali caratteristiche ha.
- **Asserzioni statiche**: si creano gli assert (controlla che le cose vadano bene durante l'esecuzione). Ci sono cose che però sono necessariamente fattibili solo a runtime.

Codice mantenibile

Esempio – Scheda Prepagata

Testo:

```
class Scheda_Prepagata {
 public:
   enum Tipo_Scheda { PAGA_DI_PIU, COSTO_RANDOM, PAGA_LA_MAMMA };
   Tipo_Scheda tipo_scheda() const;
   void addebita_chiamata(const Chiamata& call);
   // ...
 private:
   void fai_la_cosa_giusta_1(const Chiamata& call);
   void fai_la_cosa_giusta_2(const Chiamata& call);
   void fai_la_cosa_giusta_3(const Chiamata& call);
   // ...
};

void Scheda_Prepagata::addebita_chiamata(const Chiamata& call) {
   switch (tipo_scheda()) {
   case PAGA_DI_PIU:
     fai_la_cosa_giusta_1(call);
     break;
   case COSTO_RANDOM:
     fai_la_cosa_giusta_2(call);
     break;
   case PAGA_LA_MAMMA:
     fai_la_cosa_giusta_3(call);
break; }
}
```

Osservazioni:

- Questo codice rischia di generare tanti switch che occupano spazio inutile e che possono fare anche controlli inutili.
- Questo codice (come tutti) deve durare per molto tempo.
- Se dovessi fare molte modifiche, sarebbe difficile gestirle (è chiuso alle modifiche).

MA, se ho pochi utilizzi e poche (o nessuna) modifica da fare, allora va bene.

- L'ideale è scrivere il codice una volta sola e se cambia qualcosa non devo cambiare il codice che ho già, ma aggiungere un nuovo pezzo che funzionerà sulla base (la quale resta sempre fissa).

Miglioramenti del codice:

File – Scheda_Prepagata

```
class Scheda_Prepagata {
 public:
   virtual std::string nome_scheda() const = 0;
   virtual void addebita_chiamata (const Chiamata& call) = 0;  //"virtual" e "=0" sono aggiunti per renderlo giusto; virtual, in automatico, chiede il tipo dell'oggetto passato e se scopre che appartiene ad una classe derivata, userà il metodo della derivata oppure =0. Il tutto a RUNTIME.
   //...
}
```

File – paga_di_piu

```
class Paga_di_piu : public Scheda_Prepagata {
public:
  virtual std::steing nome_scheda() const {
    return "Paga_di_piu";
  }
  virtual void addebita_chiamata (const Chiamata& call) { //meglio mettere virtual anche se qui non è necessario
    fai_la_cosa_giusta_1(call);
  }
private:
  void fai_la_cosa_giusta_1 (const Chiamata& call);

}
```

File - main

```
#include "Scheda_Prepagata.hh"
void foo(const std::vector<scheda_prepagata*> schede, Chiamata& call) {
  for (Scheda_Prepagata* scheda_ptr : schede) {
    scheda_ptr -> addebita_chiamata(call);
  }
}
```

- (senza il "virtual" è) SBAGLIATO perché nel file main io chiamo la funzione "addebita_chiamata" non delle classi derivate ma della classe madre che non sa quale usare. Serve aggiungere virtual.
- Virtual permette di ridefinire dei metodi della classe madre tramite classi derivate separate che non modificano il codice di base (della classe madre).
Questa si chiama risoluzione dell'OVERRIDING (si applica dopo aver notato con l'OVERLOADING che è virtual).

Ricapitoliamo:
Creiamo una gerarchia di classi con tutte le possibili concretizzazioni (classi derivate) della classe madre dicendo che ereditano pubblicamente i metodi della madre, definendoli.
Abbiamo due tipi:
1) Supporto statico: l"addebita_chiamata" dentro al for del main
2) Supporto a tempo di esecuzione: nella dichiarazione del vector; interroga il valore concreto chiedendo com'è (cioè, quale figlia rappresenta).
- In caso di modifica devo solo ricollegare con il linker la nuova scheda (classe figlia) alla madre (senza ricompilare il codice che già esiste).
- Nella pratica è più difficile perché nella realtà i cambiamenti sono imprevedibili e se fosse necessario cambiare interfaccia (classe madre) allora bisognerebbe modificare tutto (comunque così si adegua molto bene).
Alla fine si avranno tanti file: 1 per la classe base e un paio per ogni concretizzazione (anche se sono tanti, non importa perché una volta creati non si dovranno più modificare).

Polimorfismo dinamico

Due aspetti:
- Tecnico: regole del linguaggio (del c++).
- Progettuale: polimorfismo dinamico applicabile ad altri linguaggi object oriented.

COME funziona

Esempio:

```
#include <iostream>

class A {
public:
  int foo() {
    std::cout << "A::foo()" << std::endl;
  }
};

class Base {
public:
  int a;
  virtual int foo() {
    std::cout << "Base::foo()" << std::endl;
  }
};

class Derived : public Base {
public:
  int a;
  virtual int foo() { // Fa l'overriding perché ha la stessa segnatura (cioè, se questa “foo” richiedesse un intero, sarebbe diversa e nasconderebbe sempre quella di Base; “foo” della classe derivata per sfruttare virtual deve essere uguale a quella di base nel nome, numero e tipo dei valori anche quello implicito quindi se una “foo” è const anche l'altra deve esserlo)

    Base* base = static_cast<Base>(this); // Static_cast non è necessario
    base -> foo();  //Queste due righe generano una chiamata infinita quindi non vanno fatte

    foo(); //Se scrivessi così avrei ricorsione infinita

    Base::foo(); //Risolve i problemi precedenti per chiamare la “foo” di Base
    std::cout << "Derived::foo()" << std:: endl;
  }
};

Base* dammi_un_B() {
  return new Derived();  //La conversione da “puntatore a Derived” a “puntatore a Base”, si può fare
}
```

```
int main() {
  std::cout << sizeof(A) << std::endl;  // 4 byte
  std::cout << sizeof(B) << std::endl;  // 16 byte; Questo perché è subentrato qualcosa che chiede di che tipo è veramente la classe (puntatore di 8 e padding); supporto fornito da RTTI (runtime  type identification) che può essere gestito dall'utente che appunto interroga i byte in più per conoscere il tipo vero della classe.
// Quindi: creando una classe dinamica, da qualche parte viene creato un oggetto che contiene dati riguardanti quella classe (e un puntatore che punta ad esso).

  A a;

  a.foo();
  Base b;
  b.foo();  // Un compilatore furbo esegue subito il codice della classe Base, senza fare ricerche

  Base* pb = dammi_un_B();

  if (Derived* pd = dynamic_cast<Derived*> (pb)) {
    //usa pd
  }
// ”pb” è della classe Derived? se si, metti l'indirizzo del pb in pd altrimenti restituisci un puntatore nullo per dire che non hai pb
// In linea di principio questo è male perché ciò è una violazione della astrazione (questo in pratica costringe l’utilizzo dello switch che però come abbiamo visto è male; è qualcosa che abbiamo cercato di rimuovere).

  Derived& pd = dynamic_cast<Derived&>(*pb);  // Il riferimento & non può essere nullo; se pb è della classe Derived ok, ma se non lo è cerca di dare un indirizzo nullo al & e quindi lancia un eccezione. Questa riga di codice non si vede praticamente mai.

  pb -> foo();  // Fa in modo che il supporto a tempo di esecuzione vada a controllare i byte supplementari per capire di che tipo è l'oggetto passato; in base a cosa punta pb, verrà eseguito o un codice o un altro.
// Abbiamo un metodo “foo” marcato virtuale; viene invocato tramite puntatore; il tipo statico e quello dinamico non coincidono; nella classe Derived qualcuno ha fatto l'override.
// Invoca quella di Derived.

  Base* pb = dammi_un_B();
  pb -> Base::foo();  // Invoca forzatamente la “foo” di Base

  return 0;
};
```

Override:
1) mi stanno invocando tramite un puntatore o riferimento --> vado avanti con i passaggi successivi, altrimenti il tipo statico e dinamico sono identici e quindi so già quale funzione chiamare.
2) in questo particolare punto del codice il tipo statico è diverso dal dinamico --> se si, vado avanti con il passaggio successivo, altrimenti termino.
3) qualcuno tra le derivate ha fatto l'overriding --> se si, vince l'ultimo (delle derivate) a fare l'overriding.
- Se era chiamata con "Base::foo();" allora possiamo già sapere quale chiamare.
Overiding: la selezione del codice che viene eseguito per una certa chiamata avviene a tempo di esecuzione usando un puntatore.
Il puntatore accede ad un oggetto che contiene una piccola parte di memoria in più, che indica per ogni metodo virtuale qual è il codice da eseguire.
Questa informazione in più può essere usata anche per farsi dire "quale animale sei?".

Virtual
Quali metodi possono essere dichiarati virtuali e perché?
- NO costruttori: perché l'oggetto deve essere già stato creato per potergli domandare di che tipo è.
- NO metodi statici (metodi che lavorano sulla classe e non sull'oggetto; non hanno il this): non avendo il this non hanno modo di chiedere il tipo dell'oggetto.
- SI distruttori (tranne casi rarissimi): grazie a virtual possiamo invocare il distruttore giusto a seconda di ciò che devo eliminare.
- SI metodi dinamici.
- "NO" metodi templatici: sarebbe SI ma chi ha implementato il C++ ha deciso di NO. Nel virtual c'è il puntatore che indica appunto il tipo della funzione ma la funzione templatica non è una funzione è uno schema che genera tante possibili funzioni e quindi genererebbe una tabellina dinamica che varia di dimensione arbitraria (molto difficile da gestire).
- Metodi virtuali puri: dichiarati ma non implementati. Esempio: virtual foo() = 0;
"=0" è un suggerimento; 0 è il puntatore nullo cioè non c'è il puntatore alla funzione; nella tabellina avrò 0 per quella funzione.

La **classe è astratta** se: ha metodi virtual puri perché mancano pezzi di implementazione.
Altrimenti è una **classe concreta**.
Classe mista se l'implementazione dei virtual puri viene fatta dalle derivate.

Metodo puro: non ha implementazioni (il distruttore di una derivata deve avere sempre dentro il distruttore della classe madre; il distruttore di una classe derivata è sempre implementato).

ESERCIZIO – risoluzione overloading (semplice) e overriding
Indicare l'output del seguente programma:

```
#include <iostream>

class Base {
public:
   Base() {
      std::cout << "Constructor Base::Base()" << std::endl;
   }
   virtual void f(int) {
```

```
        std::cout << "Base::f(int)" << std::endl;    }
    virtual void f(double) {
        std::cout << "Base::f(double)" << std::endl;
    }
    virtual void g(int) {
        std::cout << "Base::g(int)" << std::endl;
    }
    virtual ~Base() {
        std::cout << "Destructor Base::~Base()" << std::endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        std::cout << "Constructor Derived::Derived()" << std::endl;
    }
    void f(char c) {
        std::cout << "Derived::f(char)" << std::endl;
    }
    void g(int) {
        std::cout << "Derived::g(int)" << std::endl;
    }
    ~Derived() {
        std::cout << "Destructor Derived::~Derived()" << std::endl;
    }
};

int main() {
    1:      Base b;
    2:      Derived d;
    3:      Base& rb = b;
    4:      Base* pb = &d;
    5:      std::cout << "=== 1 ===" << std::endl;
    6:      b.f(1);
    7:      rb.f('a');
    8:      rb.g(1);
    9:      std::cout << "=== 2 ===" << std::endl;
    10:     d.f(1);
    11:     d.f(1.0) ;
    12:     d.g(3.3) ;
    13:     std::cout << "=== 3 ===" << std::endl;
    14:     pb->f(1.0);
    15:     pb->f('a');
    16:     pb->g(3.3);
    17:     return 0;
}
```

Outoput (soluzione):

```
1:    Constructor Base::Base()
2:    Constructor Base::Base()
      Constructor Derived::Derived()
=== 1 ===
6:    Base::f(int)
7:    Base::f(int)
8:    Base::g(int)
=== 2 ===
10:   Derived::f(char)
11:   Derived::f(char)
12:   Derived::g(int)
=== 3 ===
14:   Base::f(double)
15:   Base::f(int)
16:   Derived::g(int)
17:   Destructor Derived::~Derived()
      Destructor Base::~Base()
18:   Destructor Base::~Base()
```

Spiegazione:

1: Si usa il costruttore di Base.
2: Il costruttore di Deriver prima usa il costruttore di Base.
3-4: Il & e il * non costruiscono niente (non stampano niente).
In * il tipo statico è Base quello dinamico è Derived (si può fare perché Derived deriva da Base).
6: Cerco "f" dentro Base; ne ho due; invoco la 1; è indipendente se ho un virtual o no.
7: "rb" è un riferimento ma tipo statico e dinamico non sono diversi (è migliore la int piuttosto che il double; non vediamo la candidata dentro Derived).
8: Stessa cosa di prima solo che ho una sola candidata.
10-12: "d" è un oggetto Derived (di fatto, non ho da fare risoluzione di overriding quindi risolvo solo l'overloading).
10: Ho una sola candidata, le altre (in Base) sono nascoste.
11: Stessa cosa della 10 (con conversione).
12: Stessa cosa della 10.
14-16: "pb" è un puntatore con tipo statico Base e tipo dinamico Derived (applico overriding).
14: Cerco nella classe Base (perché puntatore a Base); la migliore è double (quella che vuole chiamare il compilatore) PERÒ per overriding devo controllare se in Derived c'è una funzione uguale alla Base, ma non c'è (c'è ma NON con parametro double)
15: (Cerco in Base) vince quella con int, faccio overriding ma non trovo niente di valido in Derived.
16: (Cerco in Base) ce né solo una; faccio overriding e scopro che in Derived c'è una funzione che coincide e quindi questa sarà quella utilizzata.
17: Il return 0; richiama il costruttore che distrugge in ordine inverso gli oggetti che sono stati creati quindi:
Per "d": chiama il distruttore di Derived e subito dopo chiama il distruttore di Base.
Per "b": invoca il distruttore della classe Base.

FINE Esercizio

ESERCIZIO – Indipendenza del codice

Testo:

Uno strumento software utilizza codice come il seguente allo scopo di gestire la produzione automatica di documentazione secondo diversi formati di stampa.

```
#include <string>
 using std::string;
 class Manual_Generator {
 public:
    virtual void put(const string& s) = 0;
    virtual void set_boldface() = 0;
    virtual void reset_boldface() = 0;
    // ...
 };
 class HTML_Generator : public Manual_Generator {
 public:
    void put(const string& s);
    void set_boldface();
    void reset_boldface();
    void hyperlink(const string& uri, const string& text);
    // ...
 };
 class ASCII_Generator : public Manual_Generator {
 public:
    void put(const string& s);
    void set_boldface();
    void reset_boldface();
    void page_break();
    // ...
 };

 void f(Manual_Generator* mg_p) {
    // ...
    HTML_Generator* html_p = dynamic_cast<HTML_Generator*>(mg_p);
    if (html_p)
       html_p->hyperlink("http://www.cs.unipr.it/ppl", "PPL");
    else
       mg_p->put("PPL (http://www.cs.unipr.it/ppl)");
    // ...
    ASCII_Generator* ascii_p = dynamic_cast<ASCII_Generator*>(mg_p);
    if (ascii_p)
       ascii_p->page_break();
    else
       // Nota: usare il tag HR per simulare il cambio pagina in HTML.
       mg_p->put("<HR>");
    // ...
 }
```

Svolgimento:
L'obbiettivo è scrivere "f" in modo indipendente dal "nuovo generatore di manuali" che aggiungerò (con gli if e else non si può perché essi pretendono di sapere tutti i tipi di sottoclasse esistenti).
Problemi da risolvere:
- Nella "f" si chiede (con if) qual è l'oggetto che sta dietro.
- Nella classe astratta non ci sono i virtual.

Codice corretto:
```
//In generale, servono anche le guardie che per comodità non scriverò
#include <string>
using std::string;

//FILE Manual_Generator.hh
  class Manual_Generator {
  public:
    virtual void put(const string& s) = 0;
    virtual void set_boldface() = 0;
    virtual void reset_boldface() = 0;
    virtual void hyperlink(const string& uri, const string& text);
    virtual void page_break();
    // ...
};
////////////////////////////////////////////

//FILE HTML_Generator.hh
#include "Manual_Generator.hh"
  class HTML_Generator : public Manual_Generator {
  public:
    virtual void put(const string& s);
    virtual void set_boldface();
    virtual void reset_boldface();
    virtual void hyperlink(const string& uri, const string& text);
    virtual void page_break() {
      put("<HR>");
    }
    // ...
};
////////////////////////////////////////////

//FILE ASCII_Generator.hh
#include "Manual_Generator.hh"
  class ASCII_Generator : public Manual_Generator {
  public:
    virtual void put(const string& s);
    virtual void set_boldface();
    virtual void reset_boldface();
    virtual void page_break();
    virtual void hyperlink(const string& uri, const string& text) {
```

```
      put(text);
      put(" (");
      put(uri);
      put(")");
    }
...
};
//////////////////////////////////////////

//FILE PDF_Generator.hh
//Aggiunto per mostrare che posso aggiungere una nuova classe senza problemi
#include "Manual_Generator.hh"
  class PDF_Generator : public Manual_Generator {
  public:
    virtual void put(const string& s);
    virtual void set_boldface();
    virtual void reset_boldface();
    virtual void page_break();
    virtual void hyperlink(const string& uri, const string& text);
...
};
//////////////////////////////////////////

//FILE client.cc
#include "Manual_Generator.hh" //Basta includere solo questo senza conoscere le classi derivate
  void f(Manual_Generator* mg_p) { //Non verrà mai modificata tramite l'aggiunta di nuove classi
    // ...
      mg_p -> hyperlink("Http://...");
    // ...
      mp_g -> page_break();
    // ...
}
//////////////////////////////////////////
```

FINE Esercizio

Ricapitoliamo:

- Definire la classe astratta (Base) è la cosa più importante che permette di separare due mondi che evolvono diversamente (tra cui, quello dei programmatori che possono inventare tanti Manual_Generetor).
E se io mi dimentico qualcosa nell'interfaccia, tutti gli sviluppatori che hanno aggiunto delle implementazioni dovranno cambiarle.
- Avendo pochi dettagli implementativi (molto astratta) le cose restano stabili.

Make file

In un esempio di fattoria (con eseguibile “Fattoria”, classe base “animale”, classi derivate “cane, gallina, gallo, mucca” e una funzione “canzone” che utilizza la classe astratta “animale”) ci sono molti file sorgente e un certo numero di dipendenze logiche nel programma (“gallo” dipende dalla classe “animale”; la canzone invece dipende solo dall'animale astratto; il maker invece dipende da tutti) queste dipendenze si notano quando si vuole costruire o ricorstruire l'eseguibile cioè ricompilare qualcosa in base alle dipendenze logiche.
Si usano **Make file** con un comando make che tiene traccia delle dipendenze.
Vediamo com'è il make file (non è necessario farlo, è solo per capire il concetto)

```
//Make file
HEADER_ANIMALI = \
Cane.hh
Gallina.hh
Gallo.hh
Mucca.hh
CC = $(CXX)
CXXFLAGS = -W -Wall -g

Fattoria: Fattoria.o Maker.o Canzone.o Animale.o \ Cane.o Mucca.o Gallo.o Gallina.o

Fattoria.o: Fattoria.cc Canzone.hh Maker.hh Animale.hh

Canzone.o: Canzone.cc Canzone.hh Animale.hh

Maker.o: Maker.cc Maker.hh Animale.hh $(HEADER_ANIMALI)

Cane.o: Cane.cc Cane.hh Animale.hh
Gallina.o: ...
Gallo.o: ...
Mucca.o: ...

clean:
	-rm -f *.o Fattoria
//FINE Make file
```

- Questo codice aggiorna la “Fattoria.exe” cioè va a vedere l'ultima modifica su “Fattoria.exe” se è minore della data di ultima modifica del maker; se non è minore allora non devo aggiornare.
- Se modifico “Mucca.cc” esso vede che deve modificare “Mucca.o”; ricorsivamente nota che poi deve modificare anche “Fattoria.exe” (ricompilo solo quello che serve).

- Dobbiamo sapere bene quali sono le dipendenze (per evitare compilazioni inutili).
- Dobbiamo sapere cosa includere (includendo cose che non servono si ha un rallentamento nella compilazione che può essere anche molto dispendioso).

- Nel mondo del lavoro ci sarà uno strumento simile al maker che terrà traccia delle dipendenze. (c'è un modo per fare il make molto più compatto; ci sono anche dei pacchetti che permettono di fare il make in automatico).

Trovare memory leek

Usando come esempio la fattoria, questo è un esempio di main:

```
#include
int main () {
  std::vector<Animale*> va;
  popola_fattoria(va);
  canta_canzone(std::cout, va);
  //svuota_fattoria();
  return 0;
}
```

- Il problema è trovare i memory leek.

Esiste uno stromento free per fare ciò: valgrind (è una suite per capire cosa fanno i programmi).

Vagrind tiene traccia di ogni possibile utilizzo di memoria per trovare eventuali memory leek (non necessariamente funziona sempre).

- Usandolo su Fattoria scopro che ci sono 32 bytes persi (memory leek)

Non abbiamo rilasciato la memoria dichiarata con "new (unsigned long)" (esempio di errore).

Bisogna togliere dal commento lo "svuota.fattoria()" perché con esso si risolve la deallocazione.

- Proviamo ad implementare NON virtualmente il costruttore di Animale:

Il compilatore da Warming ma valgrind no... questo perché valgrind dice solo se un programma ha o no memory leek (in questo caso, sono fortunato perché gli animali non hanno bisogno di creare niente; stampano e basta ma se creassero qualcosa, allora avrei errore anche da valgrind).

SOLID (SRP – OCP – LSP – ISP – DIP)

In progettazione object oriented SOLID indica dei principi "solidi" che se rispettati permettono di scrivere programmi nel modo "migliore".

SOLID è un acrostico cioè ogni lettera indica l'iniziale di un altro acronimo.

S – SRP (single responsibility principle)

(Wiki - principio di singola responsabilità): *afferma che ogni classe dovrebbe avere una ed una sola responsabilità, interamente incapsulata al suo interno.*

Ogni entità software (classe, funzione, etc) costruita, dovrebbe avere una sola responsabilità; dovrebbe far bene una sola cosa.

Questo perché è più semplice e debuggabile e anche perché si può modificare facilmente.

Il SRP si applica per evitare di fare troppe cose insieme perché altrimenti:

- L'implementazione si complica.
- La documentazione e l'uso diventa complesso.
- Il ri-uso è faticoso.
- Costi manutenzione del codice più elevati.
- Aumenta il codice duplicato.

O – OCP (open/closed principle)

(Wiki - principio aperto/chiuso): *un'entità software dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche.*

Le entità del software dovrebbero essere aperte alle estensioni ma chiuse alle modifiche (ottenuto negli esempi con virtual).

Client-Server diretto: non accetta modifiche.

Client-Abstract Server-Server: è l'interfaccia che gestisce le modifiche.
È il più importante e **comprende anche il DIP.**

L – LSP (Liskov substitution principle)

(Wiki - principio di sostituzione di Liskov): *gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del programma che li utilizza.*
S è un sottotipo di T.
Se q(x) è una proprietà valida per oggetti x di tipo T.
Allora q(y) è una proprietà valida per oggetti y di tipo S.
In pratica, una proprietà della classe base DEVE valere anche per le derivate.
Cosa vuol dire che una certa classe D eredita dalla classe principale B?
Significa che noi ci aspettiamo un certo comportamento da D.
Se abbiamo un oggetto di tipo D e di tipo B e chiedo un operazione che sa fare B, entrambi gli oggetti avranno lo stesso comportamento; nel caso descritto vale LSP, altrimenti no.

Esempio:
Ho l'oggetto Rettangolo e serve creare anche l'oggetto Quadrato.
Il matematico dice "i quadrati sono rettangoli con proprietà in più" quindi:
Quadrato (derivata) → Rettangolo (base)
Rettangolo: area, perimetro.
Quadrato: usa i metodi di rettangolo.
- Aggiungo un po' di metodi:
Rettangolo: get base, get height
Quadrato: può usare quelli di Rettangolo
- Aggiungo:
Rettangolo: set Base, set Height
Quadrato: posso usare i metodi di rettangolo ma sbaglio perché io devo poter sostituire il quadrato in un qualsiasi programma dove ho usato rettangolo e deve funzionare uguale.
Esempio dove non funziona:

```
void foo( Rettangolo &R) {  //passo oggetto Rettangolo
  double a = R.area();  //a = area
  R.setHeight (r.getHeight()*2);  //moltiplico altezza per 2
  double a2 = R.area();  //a2 = 2*area = 2*a
  assert( 2*a == a2 );
}
```

Il Quadrato moltiplicando l'altezza per 2, raddoppia anche la base e quindi l'area sarà 4*a e non 2*a.
(qui l'aspetto sintattico era soddisfatto ma non la LSP).
I metodi hanno un contratto e in questo caso è stato violato.
- Qual'è il contratto? E dove non si rispetta?
"area" funziona ma c'è una condizione nel Rettangolo che dice "se modifico l'altezza NON devo modificare la base" nel Quadrato invece modificando l'altezza si modifica anche la base (violazione).
Quindi nell'esempio precedente i quadrati non sono rettangoli.
Il rettangolo doveva essere statico (non doveva cambiare).
Contratto: pre-condizione E (invariante) → post-condizione E (invariante)
Diminuendo le richieste della pre-condizione, non ci sono problemi (se invece si aumentano le richieste, non va bene).

Relazioni IS-A, HAS-A

Facendo riferimento all'automobile:

IS – A:

Mi interessa dire che l'automobile E' un mezzo di trasporto (ma non ha un mezzo di trasporto)? Uso la relazione IS-A.

- Se non ci sono parti del codice che lo richiedono allora non faccio la relazione IS-A (qui è importante sapere cosa vuole l'utente).
- Se la faccio, creo il mezzo di trasporto astratto sfruttando in genere l'ereditarietà.

HAS – A:

Mi interessa dire che l'automobile HA 4 ruote (ma non è una ruota)? Uso la relazione HAS-A.

- Si usa se ti metti a lavorare con oggetti concreti.
- Se hai HAS-A non serve l'ereditarietà; si preferisce il contenimento o composizione di oggetti interni alla classe.

Situazione strana: voglio essere sicuro che un certo sottoggetto deve essere il primo ad essere costruito; è meglio usare l'ereditarietà.

Esempio - Relazioni

1° metodo)

```
class Basic_Protocol {  //non usabile perché è solo un raccoglitore astratto
  //...
};

class Protocol_A : /*piblic*/ private Basic_Protocol {  //dà la possibilità di convertire un oggetto di Protocol_A in Basic_Protocol e non va bene; A non è un protocollo Base (NON è ISA). Invece di public metto private.
  //...
};
```

2° metodo)

```
class Protocol_A { //Composizione
private:
  Basic_Protocol bp;
public:
  //...
}
```

Il 2° metodo è meglio del precedente perché:

- aggiungere elementi (quante ruote ha l'auto? Nel 1° puoi averne solo 1, nella seconda puoi creare 4 oggetti ruota).
- nel ° si può confondere il tipo Protocol_A con Basic_Protocol (sbagliato).
- si usa il 1° per pigrizia perché se io ho 4 metodi in Basic_ Protocol, nel 1° scrivo meno mentre.

Confronto lunghezza codice (ipotizzando di avere i metodi f1, f2 in Basic_Protocol e di doverli usare in Protocol_A):

1° metodo)

```
class Protocol_A : private Basic_Protocol { //Composizione
public:
  using Basic_Protocol::f1; //Per essere sicuro che nel blocco verrà usata la f1 di Basic_Protocol
  {
     f1(...);
     //...
  }
  //...
}
```

2° metodo)

```
class Protocol_A { //Composizione
private:
  Basic_Protocol bp;
public:
  void f1() {
    bp.f1();
    //...
  }

  void f2() {
  {
    bp.f2();
    //...
  }
  //...
}
```

- Il contenimento è sempre la migliore (anche se bisogna scrivere di più soprattutto se i metodi “f” sono molti e non solo 2).

Capire quando si è in relazione ISA (IS - A) o HASA (HAS - A):
bisogna mettersi nei panni dell'utente per capire cosa gli interessa.
Es: B (classe base), D1 e D2 (classi derivate da B)
- ISA: se interessano solo le classi derivate è meglio l'ereditarietà multipla (1° metodo); l'interfaccia resta costante (cambiano le concretizzazioni come nell’esempio della fattoria).
- HASA: viene fornito solo un pezzo dell'implementazione pronto per le varie concretizzazioni e riutilizzabile quindi in vari contesti.

I – ISP (Interface segregation principle)

(Wiki - principio di segregazione delle interfacce) *Sarebbero preferibili più interfacce specifiche, che una singola generica.*
ISP: è un sottoinsieme del SRP.

ESERCIZIO 3 (uso ISP) - 27/2/2006

Un'applicazione software si interfaccia ad un'apparecchiatura di comunicazione multi-funzione (fax e modem) mediante il corrispondente driver proprietario, utilizzando un file header come il seguente:

```
// File FaxModem.hh
class FaxModem_AllStars {
private:
  // ...
public:
  void fax_function_1();
  void fax_function_2(const char*);
  void fax_function_3(int);
  // ...
  void modem_function_1(const char*);
  void modem_function_2();
  void modem_function_3(unsigned int);
  // ...
};
```

Il codice utente, nel quale le due funzionalità dell'apparecchiatura sono sempre utilizzate in contesti distinti, è il seguente:

```
// File User.cc
#include "FaxModem.hh"
void user_function_1(FaxModem_AllStars& f) {
  f.fax_function_1();
  // ...
  f.fax_function_3(12);
  // ...
}
void user_function_2(FaxModem_AllStars& m, const char* command) {
  m.modem_function_1(command);
  // ...
  m.modem_function_3(1024);
  // ...
}
void user_function_3(FaxModem_AllStars& f, FaxModem_AllStars& m) {
  f.fax_function_2("+390521906950");
  // ...
  m.modem_function_2();
// ...
}
```

L'utente vuole eliminare la dipendenza del proprio codice dal produttore AllStars, in quanto sul mercato esistono altri produttori di apparecchiature multifunzione analoghe, non chè produttori di apparecchiature che supportano una sola delle due funzionalità.

Mostrare le modifiche da apportare in tal senso al codice utente, comprese eventuali nuove interfacce. In particolare, mostrare come il driver proprietario di AllStars (non modificabile dall'utente) possa essere interfacciato con il nuovo codice utente.

Risolvere INTERFACCIA

- Eliminare le dipendenze da Allstar:

Dipendenza attuale:

user.cc → FM_AS (dipendenza da invertire)

Risolvo con:

user.cc → FaxModem.hh ←(IS-A)— FM_AS

←(IS-A)— FM_USR //Questo è un esempio per mostrare il collegamento se dovessi aggiungere nuovi tipi di fax

- Bisogna usare adattatori per evitare di modificare FM_AS

user.cc → FaxModem.hh ←(IS-A)— Adapter_FM_AS —(HAS-A)→ FM_AS

←(IS-A)— Adapter_FM_USR —(HAS-A)→ FM_USR

Finora però non è servito il ISP.

- "Apparecchiature che supportano solo una delle funzionalità" (Es: la Tim mi da un modem che non sa fare il Fax)

user.cc → FaxModem.hh ←(IS-A)— Adapter_FM_AS —(HAS-A)→ FM_AS

←(IS-A)— Adapter_FM_USR —(HAS-A)→ FM_USR

←(IS-A)— Adapter_M_TIM —(HAS-A)→ M_TIM //non sa fare il Fax

Abbiamo una classe astratta che pretende troppo dalle sue implementazioni (FaxModem pretende che tutti i modem sappiano fare il fax).

“Fax” e “Modem” in realtà sono due implementazioni da separare.

ISP: se si può suddividere l’interfaccia in porzioni più piccole meglio farlo.

user.cc → Fax.hh ←(IS-A)— Adapter_FM_AS —(HAS-A)→ FM_AS

Modem.hh ←(IS-A)–/

←(IS-A)— Adapter_FM_USR —(HAS-A)→ FM_USR

←(IS-A)–/

←(IS-A)— Adapter_M_TIM —(HAS-A)→ M_TIM //Eredita solo dal Modem

Così è più facile capire l'interfaccia.

FM_AS e FM_USR usano l’ereditarietà multipla (due frecce IS-A) su Fax e Modem.

MODIFICA del CODICE originale

```
// File FaxModemAS.hh  ----------------------
class FaxModemAS {
private:
  // ...
public:
  void fax_function_1();
  void fax_function_2(const char*);
  void fax_function_3(int);
  // ...
  void modem_function_1(const char*);
  void modem_function_2();
  void modem_function_3(unsigned int);
  // ...
};
//--------------------------
```

```
//File Fax.hh  ----------------------
class Fax {
  private:
    // ...
  public:
    virtual void fax_function_1() = 0;
    virtual void fax_function_2(const char*) = 0;
    virtual void fax_function_3(int) = 0;
    virtual ~Fax();
    // ...
};
//--------------------------

//File Modem.hh  ----------------------
class Modem {
  private:
    // ...
  public:
    virtual void modem_function_1(const char*) = 0;
    virtual void modem_function_2() = 0;
    virtual void modem_function_3(unsigned int) = 0;
    virtual ~Modem();
    // ...
};
//--------------------------

//Adapter_FM_AS.hh  ----------------------
#include "Fax.hh" //interfaccia
#include "Modem.hh" //interfaccia
#include "FaxModemAS.hh" //implementazione

class Adapter_FM_AS : public Fax, public Modem { //Ereditarietà multipla
  private:
    FaxModemAS* p_fmas;
  public:
      explicit Adapter_FM_AS (FaxModemAS* p) : p_fmas(p) {
      }

      virtual void fax_function_1() {
        p_fmas -> fax_function_1();
      }
      virtual void fax_function_2(const char* s) {
        p_fmas -> fax_function_2(s);
      }
      virtual void fax_function_3(int n) {
        p_fmas -> fax_function_3(n);
      }
```

```
    virtual void modem_function_1(const char* a) {
      p_fmas -> modem_function_1(a);
    }
    virtual void modem_function_2() {
      p_fmas -> modem_function_2();
    }
    virtual void modem_function_3(unsigned int b) {
      p_fmas -> modem_function_3(b);
    }

    virtual ~Adapter_FM_AS() {}
}
//-------------------------

// File User.cc  ---------------------
#include "Fax.hh"
#include "Modem.hh"
#include "FaxModemAS.hh"
 void user_function_1(Fax& f) {
   f.fax_function_1();
   // ...
   f.fax_function_3(12);
   // ...
 }
 void user_function_2(Modem& m, const char* command) {
   m.modem_function_1(command);
   // ...
   m.modem_function_3(1024);
   // ...
 }
 void user_function_3(Fax& f, Modem& m) {
   f.fax_function_2("+390521906950");
   // ...
   m.modem_function_2();
   // ...
 }
//-------------------------
```

FINE Esercizio.

D – DIP (Dependency inversion principle)

(Wiki - principio di inversione delle dipendenze) *Una classe dovrebbe dipendere dalle astrazioni, non da classi concrete.*
I moduli di alto livello non devono dipendere da quelli di basso livello. Entrambi devono dipendere da astrazioni.
Le astrazioni non devono dipendere dai dettagli; sono i dettagli che dipendono dalle astrazioni.
Policy Layer → Mechanism Layer → Utility Layer

Questa dipendenza implica che cambiando qualcosa in Mechanism Layer devo poi modificare Policy Layer.
Quindi faccio:
Policy Layer → Mechanism Interface ← Mechanism Layer → Utility Layer Interface ← Utility Layer.
Per fare ciò bisogna individuare le dipendenze (esistono file su internet che permettono di fare ciò).
- non bisogna esagerare perché comunque qualche dipendenza ci sarà sempre.
- bisogna determinare quali dipendenze lasciare e quali togliere perché ad ogni astrazione fatta, il codice si complica.

L'ottimizzatore spesso fa l'espansione inline della funzione ma nei virtual no, quindi c'è poca ottimizzazione.
Il DIP però è un aspetto solo sintattico (scritto) ma a noi interessa anche il lato semantico (senso).

ESERCIZIO 4 (uso OCP , DIP) - 20/9/2005

Una ditta di sviluppo software si occupa della manutenzione di un'applicazione che, per implementare alcune sue funzionalità, utilizza la libreria "BiblioSoft", fornita da terzi e per la quale non si ha a disposizione il codice sorgente. In particolare, l'applicazione accede alla classe BSoft, la cui interfaccia sarà mostrata in seguito.
L'utilizzo che né viene fatto della classe BSoft, all'interno dell'applicazione, è tipicamente il seguente (in arancione perché nello svolgimento lo modificherò):

```
void f(BSoft& x, const BSoft& y, int n)
{  // ...
   if (n > 0)
     x.s3(n);
   else {
     x.s1();n = 5;
   } // ...
   x.s2(y, n); // ...
}

//NON si può modificare (vale anche nella soluzione)
class BSoft {
  private:
    // ...
  public:
    void s1();
    void s2(const BSoft& y, int n);
    void s3(int n);
    // ...
};
```

Sul mercato esiste un'altra libreria "BiblioWare" che offre utilità simili e che è preferita da alcuni degli utenti della ditta. L'interfaccia in questo caso è la seguente:

```
//NON si può modificare (vale anche nella soluzione)
class BWare {
  private:
    // ...
  public:
    void w1();
    void w2_1(const BWare& y);
    void w2_2(int n);
    void w3(int n);
    // ... ecc.
};
```

L'invocazione dei metodi BWare::w1() e BWare::w3(n) ha lo stesso effetto dell'invocazione di BSoft::s1() e BSoft::s3(n), rispettivamente. I metodi BWare::w2_1(x) e BWare::w2_2(n), se invocati immediatamente uno dopo l'altro (sullo stesso oggetto), hanno lo stesso effetto dell'invocazione di BSoft::s2(x, n).
Mostrare come può essere integrato/modificato il codice precedente al fine di rendere agevole l'utilizzo di una qualunque delle due librerie (Nota Bene: non è possibile modificare il codice di BSoft e BWare).

Risolvere INTERFACCIA

utente.cc → BSoft BWare
- ho BSoft che implementa servizi.
- utente.cc dipende da BSoft e da BWare.
- in futuro può essere aggiunto qualcosa.
- non posso modificare BSoft o BWare.

Per risolvere queste limitazioni, creo la classe astratta Biblio:
utente.cc —(include)→ Biblio.hh ←(IS-A)— BSoft_Impl.hh —(HAS-A)→ BSoft.hh
utente.cc —(include)→ Biblio.hh ←(IS-A)— BWare_Impl.hh —(HAS-A)→ BWare.hh
- con solo Biblio l'utente è a posto perché usa solo Biblio.
- la doppia freccia IS-A verso Biblio.hh, non è un problema.
- se potevo modificare BSoft e BWare avrei applicato una relazione IS-A diretta con Biblio.

```
//File Bliblio.hh – Classe astratta  ----------------------
class Biblio {
public:
  virtual void s1() = 0;
  virtual void s2(const Biblio& y, int n) = 0;
  virtual void s3(int n) = 0;
  virtual ~Biblio() {}
}
//--------------------------
```

- potremmo fare nel file BSoft

```
#include "Biblio.hh"
class BSoft : public Biblio {...}
```

- ma non posso modificarlo quindi faccio:

```
//FILE BSoft_Impl  ----------------------
#include "Biblio.hh"
#include "BSfot"
class BSoft_Impl : public Biblio {
  private:
    BSoft bs;
  public:
    void s1() {
      bs.s1();
    }
    void s2(const /*Biblio*/ BSoft& y, int n) { //Con Biblio NON compilerebbe perché non c'è
conversione da Biblio a BSofty
      BSoft_Impl&  y_bs = static_cast<BSoft_Impl&>(y);
      bs.s2( y_bs.bs , n); //La conversione l'ho fatta quindi so che è BSoft, MA non posso controllare
la conversione

      BSoft_Impl* y_bs_ptr = dynamic_cast<BSoft_Impl*>(&y); //Da 0 se la conversione fallisce
      if(y_bs_ptr == 0) {
        Panic_Mode();
      }
      bs.s2(y_bs_ptr , n);
    }
    void s3(int n) {
      bs.s3(n);
    }
    // ...
};
//--------------------------

 //FILE BWare_Impl  ----------------------
#include "Biblio.hh"
#include "BWare.hh"

class BWare_Impl : public Biblio {
  private:
    BWare bw;
  public:
    void w1() {
      bw.w1();
    }

    void w2_1(const BWare& y, int n) {
      BWare_Impl* y_bw_ptr = dynamic_cast<BWare_Impl*>(&y);
      if(y_bs_ptr == 0) {
        Panic_Mode();
      }
      bw.w2_1(y_bw_ptr);
    }
```

```
    void w2_2(int n) {
      bw.w2_2(n);
    }
    void w3(int n) {
      bw.w3(n);
    }
    // ...
};
//-------------------------

// Codice Utente  ----------------------
#include "Biblio.hh" //Classe astratta
void f(Biblio& x, const Biblio& y, int n) {
    // ...
    if (n > 0)
      x.s3(n);
    else {
      x.s1();
      n = 5;
    }
    // ...
    x.s2(y, n);
    //...
}
//-------------------------
```

FINE Esercizio.

ECCEZIONI (Ereditarietà)

Ci sono casi in cui l'ereditarietà è un abuso e quindi ora osserviamo dove viene utilizzata l'ereditarietà:

- **Exception**: le usavamo ma non sapevamo esattamente come funzionavano.

Osserviamo la classe exception della std: c'è un metodo virtuale e di suo all'interno non ha niente. Subito sotto c'è una estensione (bad_exception) anche questa vuota.

- **Bad_exception** si usa quando io mi aspetto di vedere una certa eccezione, ma ne vedo un'altra, in tal caso lancio la bad_exception.

Poi ci sono cose che indicano come far terminare il programma in modo standard (non ci interessa).

C'è un file std_except che descrive dei tipi di eccezione (concretizzazioni):

- **Logic_error**: passo la stringa (what() dice cosa fare (prevedibile))
- **Domain_error** (è un logic): prendo sempre una stringa, ma non fa nulla di nuovo, dice solo che si ha un errore di dominio (dominio matematico) → se faccio una divisione sui razionali, ho come dominio tutti i razionali tranne 0 e questo è domain_error.

```
//L'eccezione va lanciata per valore
void foo() {
  try {
  }
  catch (const Division_BY_Zero& z) { //Sbagliato catturare per valore perché catturiamo una copia; serve il riferimento; const se non dobbiamo modificarla ma solo leggerla
  }
}
```

Come fa l'utente a conoscere tutti i tipi di errori concreti?
L'utente vuole sapere l'astrazione (logic_error). Quindi se all'utente non interessa sapere esattamente qual è l'errore, ma piuttosto gli interessa conoscere la categoria, allora scriverò:
`catch (const std::domain_error& z) {`
Mi arriva una Divisione_by_zero: l'ereditarietà lo riconosce come domain_error ma il messaggio che stampa è quello di Divisione_by_zero.
Altre sottocategorie:
- **Invalid_argument** (è un logic): l'argomento non è valido
- **Lenght_error** (logic): troppo grande (se supera il MAX_SIZE)
- **Out_of_range** (logic): vai a cercare un valore in una posizione che un vettore non ha.
- **Runtime_error** (exception): sono identici ai logic_error (non sono facilmente riconoscibili; sono qualcosa che non si può controllare).
- **Range, overflow, underflow** (runtime)

Abbiamo quindi questa gerarchia di classi per sapere di quale categoria è un'eccezione.

```
void foo() {
  try {
  }
  catch (const std::domain_error& z) { //Se mettiamo quella generale in alto, essa non permetterà di catturare quelle sotto perché lei catturerà tutto anche le derivate
    //fai qualcosa
  }
  catch (const std::length_error& z) {
    //fai qualcosa
  }
    catch (const std::range_error& z) {
    //fai qualcosa
  }
  catch (const std::exceptione& z) {
    //fai qualcosa
  }
}
```

Ereditarietà MULTIPLA

Non esiste in tutti i linguaggi (o esiste in forma limitata).
Nel C++ non è molto limitata e può essere abusata.
Vari casi:

1) classi Base con Derivate

B1 ←(IS-A)— D1 —(IS-A (publica))→ B2
Uso buono (non presenta le successive problematiche perché non ci sono dati sparsi e l'utente può fregarsene della D1 perché esso usa le B1,2).

2) Ambiguità 1: non sono classi astratte ma hanno dentro certe funzioni e dati:
B1 ha una Foo(), una Print() e un dato d
B2 ha una bar(), una Print() e un dato d
D1 eredita da entrambe (come prima ma non pubblica).
- devo cercare Foo() quindi applico la risoluzione di overloading cercando Foo in entrambe le B e qui va bene dato che ne ho solo una.
- se faccio la stessa cosa con Print() invece ottengo AMBIGUITÀ (stessa cosa con dato d).
Il compilatore rileva questo errore SOLO eseguendo il codice e non prima.
Si può risolvere con ”static cast” o “d.B1::Print();” oppure fornire una Print() in D1 che stampa prima B1 e poi B2.
Questo avviene perché qui si vuole usare la D1 mentre prima si voleva usare B1 e B2.

3) Ambiguità 2: A ← B1 ← D1 → B2 → A
Qui c`è sempre ambiguità.
Si può risolvere dando il percorso ma non è il massimo.
Si vorrebbe fare:
D1 → B1 → A (forma a diamante)
→ B2 →
Si può fare marcando l'ereditarietà con una chiave virtual
class B1 : public virtual A {...}
class B2 : public virtual A {...}
A non è solo mio ma lo condivido con altri.
Quindi quando io chiedo di usare A, non avrò più ambiguità perché ho solo una A.
(questo si usa nella standard library per fare input output sullo stesso file senza duplicarlo).
In realtà non abbiamo ancora risolto il problema perché questa cosa comporta altri problemi…

Costruttori:

A ← B1 ← D1 → B2 → A
Costruiva da sinistra a destra costruendo in ordine A, B1, A, B2, D1 (distruzione al contrario).
In questo successivo caso invece:
D1 → B1 —(virtual)→ A (forma a diamante)
→ B2 —(virtual)→
In ordine: A, B1, A è già inizializzato quindi non so se mantenere o no l'inizializzazione precedente (non possono stabilire la precedenza).
Hanno risolto dicendo che B1 non è detto che sia lui ad inizializzare A.
Le classi virtuali vengono inizializzate dall'ultimo quindi la responsabilità si da a D1 il quale deve dire chi ha la precedenza tra B1 e B2.
L'oggetto più derivato, è lui che decide come inizializzare A: D():A(...),B1(...),B2(...) → così il B2 non inizializzerà A; ci penserà D a gestirlo.

Quando si usa la forma a Diamante, il programmatore deve prevedere tutta la forma prima di iniziare a scrivere.
Nel C++ si può fare sia l'ereditarietà multipla buona sia quella cattiva.

ESERCIZIO 2 (uso ISP ed ereditarietà multipla) - 22/2/2005

Indicare l'output prodotto dal seguente programma.

```
#include <iostream>
class ZooAnimal {
 public:
   ZooAnimal() {
     std::cout << "Constructor ZooAnimal" << std::endl;
   }
   virtual void print() {
     std::cout << "ZooAnimal::print" << std::endl;
   }
   virtual ~ZooAnimal() {}
};
class Bear : virtual public ZooAnimal {
 public:
   Bear() {
     std::cout << "Constructor Bear" << std::endl;    }
   void print() {
     std::cout << "Bear::print" << std::endl;
   }
   virtual ~Bear() {}
};
class Raccoon : virtual public ZooAnimal {
 public:
   Raccoon() {
     std::cout << "Constructor Raccoon" << std::endl;
   }
   virtual ~Raccoon() {}
};
class Endangered {
 public:
   Endangered() {
     std::cout << "Constructor Endangered" << std::endl;
   }
   void print() {
     std::cout << "Endangered::print" << std::endl;
   }
   virtual ~Endangered() {}
};
```

```
class Panda : public Endangered, public Bear, public Raccoon { // Contribuisce a determinare l'ordine del primo output
  public:
    Panda() {
      std::cout << "Constructor Panda" << std::endl;
    }
    void print() {
      std::cout << "Panda::print" << std::endl;
    }
    virtual ~Panda() {}
};
int main() {
  Panda ying_yang; //Primo output (mostrato successivamente)
  ying_yang.print(); //Usa il print di Panda
  Bear b = ying_yang; //Non stampa
  b.print(); //Niente overriding; usa il print di Bear
  ZooAnimal* pz = &ying_yang; //Non stampa
  pz->print();  //Ho sia tipo statico che dinamico quindi devo risolvere overriding; il print di ZooAnimal è virtuale quindi guardo se viene fatto e quando è fatto (l'ultimo) overriding; in caso lo trovo, lo sostituisco al print di ZooAnimal; alla fine, uso il print di Panda.
  Endangered& re = ying_yang; //Non stampa
  re.print();   //Tipo statico e dinamico diversi; la print di Endengered non è virtuale quindi uso quella e non si ha overriding
  return 0;  //Nel return si distruggono in ordine inverso a come sono stati costruiti nel main
}
```

SVOLGIMENTO:

Facciamo prima il diagramma delle classi con ereditarietà:

ZooAnimal <-v-- Beer <--- Panda (schema a diamante)
<-v-- Reccon <---
<--- Endangered <---

ZooAnimal è il primo ad essere costruito.
1) Stampa:
- costuisco ZooAnimal
- costruisco Endengered
- costruisco Beer
- costruisco Reccon
- costruisco Panda

INFO VARIE:

Doxygen genera anche un grafico che indica la gerarchia di classi.
La relazione IS-A permette la conversione in automatico.
- Una ereditarietà pubblica con soli metodi static è fatta per pigrizia cioè è fatta per ereditare senza doverle riscrivere (mezzo abuso).

qt class diagramm: ogni rettangolo è una classe (la realtà è molto complicata).
http://doc.qt.digia.com/extras/qt43-class-chart.pdf
In questo caso vanno un po' in crisi i principi che abbiamo fatto; qui l'utente può mettersi in lato astratto o concreto DIPENDE ma in genere in questo caso, serve maggiormente la concretizzazione;
Es: devo fare un programma che chiede agli oggetti come sono tramite dynamic_cast (cose "negative" in certi contesti ma non in questo).
Gerarchie così complicate sono rivolte a problemi complessi ma se a me interessa solo la parte concreta, allora non è molto complicato usarle.
Due gerarchie di questa complessità, non sono utilizzabili entrambe contemporaneamente perché sono fatte per legarvi ad esse (questione di marketing).

PPL - Classe poliedro: si fa subito una suddivisione tra quelli chiusi e quelli che possono essere sia chiusi che aperti (sui primi, le operazioni sono più veloci); entrambe suddivisioni che fanno parte di una classe contenente un flag che indica com'è il poliedro.
Qui le proprietà SOLID non valgono ma dato che una modifica difficilmente ci sarà, allora si hanno degli if-else sparsi nel codice.
Perché usare IS-A e non il contenimento? Potevano farlo ma erano pigri; il poliedro è pesante e scrivere tutte le funzioni "passa-carte" è noioso.
Dato che la classe base non può essere utilizzata, come facciamo ad obbligare l'utente ad usare una delle due suddivisioni? Facciamo i costruttori della classe base come protected (utilizzabili solo dalle derivate ma non dall'esterno).

Design patters: schemi di progettazione generali che indicano problemi che capitano spesso nel voler creare un progetto complicato.
Frequenti sono (wikipedia):
- **I pattern creazionali**: nella progettazione con polimorfismo dinamico devi utilizzare astrazioni ma esiste un punto (tanti punti) nei quali bisogna generare nuovi animali e non si possono costruire animali astratti quindi si devono anche creare animali concreti.
Questo implica che vi è una dipendenza dall'implementazione.
I partern creazionali gestiscono queste dipendenze mettendole in un unico file "object factory" che contiene gli oggetti concreti (nel caso della fattoria era maker.cc il quale doveva crearli).
Così è più facile gestire le dipendenze.
Questo però significa che serve anche una factory astratta.
Se devo costruire un pulsante potrei avere due librerie diverse che costruiscono pulsanti in modo diverso e mi serve quindi una classe astratta per la gestione.
Di oggetto che costruisce il pulsante comunque ne devo avere solo uno e quindi viene gestito anche questo.
- **adapter**: adattano tra di loro due interfacce diverse.
- **decorator**: hai un oggetto pizza che ha anche un costo e gli ingredienti ma questi oggetti vengono fuori in molti modi diversi; non possiamo immaginarci tutte le possibili combinazioni quindi consentiamo all'utente durante l'esecuzione di gestire questa cosa consentendo di comporre a piacere tutte le pizze.

Il decorator è un oggetto che eredita dalla pizza e fornisce un'interfaccia uguale a quella della pizza; al suo interno si memorizza un riferimento a puntatore che punta ad un'altra pizza; siccome il metodo che calcola la pizza è overriding, il decorator sovrascrive il metodo per calcolare l'altra pizza.
Il decorator è una pizza che però si memorizza (sovrascrive) con un'altra pizza cambiando quindi anche i metodi ad essa legata.
Il decoretor è una pizza (non concreta) nella quale se si aggiunge la mozzarella, io aumento il prezzo di un certo valore.
Al PC: voglio la finestra con un certo bordo allora creo un decorator che si applica quando creo la finestra (sono pattern).
- **state**: registra determinati stati che possono essere utilizzati per cambiare qualcosa.
Uccidi 100 mostri, ricevi una medaglia (la medaglia è un oggetto astratto che dipende dallo stato).
Il numero di ciò che cambia rispetto allo stato, non è fissato.
L'oggetto osservato ha 3 metodi (e una lista):
2 uno per aggiungere e l'altro per togliere alla lista puntatori ad oggetti che osservano lo stato.
1 quando cambia lo stato, l'osservato invoca un metodo (notified) che notifica il proprio cambiamento agli oggetti della lista. Gli osservanti decideranno poi loro cosa fare.
- **visitor**: quando la gerarchia è complicata (come quella per gestire le espressioni) dove metto i virtual?
Se per caso l'utente vuole far fare operazioni diverse e nuove?
Il visitor va ad effettuare controlli (visita) su ad esempio gli animali quindi su gallina, maiale etc
Lato negativo:
- (IL SERVER) Se la nostra astrazione non va più bene quindi se serve una modifica nei metodi, questo comporta che le classi concrete inventate finora dovranno essere modificate.
- Il visitor va in crisi quando qualcuno aggiunge una nuova classe concreta alla mia gerarchia, il visitor non sapranno analizzare la nuova classe concreta (in questo caso bisogna quindi aggiornare i visitor per renderli compatibili con le nuove classi concrete).
FINE info varie.

Costruttori virtuali

Non posso dichiarare virtuali i costruttori perché ho bisogno del this ma il virtual non ha this.
In giro però si trovano costruttori virtuali che però non sono costruttori ma si comportano come tali.
Se io volessi creare una copia concreta di animale che è astratto, come faccio?
Con la costruzione virtuale:

```
class Anmale {
public:
  virtual Animale* clone() const = 0;
};

class Cane : public Animale {
public:
  Cane* clone() const { //Se cambia qualcosa nel nome, non vale l'overriding ma qui restituisco un puntatore Cane e non puntatore ad Animale; questo è possibile se esiste la relazione IS-A (LSP) tra animale e cane con la quale avviene quindi overriding
    return new Cane(*this);
  }
}

int main() {
  Animale a;
}
```

- Questo permette di poter costruire un oggetto come prototipo di un altro oggetto (che è il *this).
Es: voglio costruire un sanbernardo:
Costruisco un Animale → Quado costruisco il cane (nasce), voglio che faccia un verso (virtual)... fa quello del Cane o del Sanbernardo (classe concreta di Cane)?
E' meglio evitare mettere metodi virtual nei costruttori (o distruttori).
Mentre è proprio sbagliato usare virtual dentro a costruttori (o distruttori) che accedono a dati dell'oggetto.

Polimorfismo

Esistono 2 forme di polimorfismo:
1) **polimorfismo statico** (es template):
- Scriviamo degli schemi di funzione con cui generare nuove funzioni.
- Molto più efficienti perché a compilazione sappiamo già chi invocare e possiamo anche ottimizzare.
- Viene generato molto codice e in certi casi (rari) si peggiorano le prestazioni perché si ha parecchio codice da caricare in più.
Per risolvere in parte questo problema in certe implementazioni della stl, ci sono operazioni che valgono per più tipi (vector, liste etc) tramite l'utilizzo di puntatori a qualcosa che sono solo loro a cambiare (questo soluzione vale anche per il dinamico).
2) **polimorfismo dinamico**:
- Utilizza classi dinamiche con funzioni astratte perché si decide a tempo di esecuzione cosa farà la funzione.
- Si fanno più controlli a tempo di esecuzione.
- Si genera meno codice.

Si sta utilizzando maggiormente lo statico (anche perché è meglio intervenire durante la compilazione).
Es: il fattoriale è meglio farlo durante la compilazione SE si hanno già i dati del fattoriale così si inserisce il numero ed è più veloce.

Nel linguaggio C++ ci sono molte costanti da scrivere e spesso sono costanti derivate (calcolate) da altre; nel 2003 non si poteva definire costanti usando altre costanti salvo in casi particolari.
Hanno inventato i **const expr**: se viene invocata una funzione con dati costanti, si calcolala a tempo di compilazione e non durante il runtime.
Asserzioni: controlla (solo) durante il debugging e funziona solo a tempo di esecuzione.
Si nota che però certe asserzioni si possono controllare durante la compilazione e quindi si sono create le **static assert**:
- se l'asserzione è violata, il programma non compila.
- non c'è distinzione tra compilazione e debugging; tutto il controllo avviene durante la compilazione.

Si sta quindi creando una separazione ma che comunque non è completa:
Linguaggi molto dinamici: scripting, java, C++ (vecchio)
Linguaggi molto statici: C++

Es:
```
class C {
  template <...>
  virtual void foo() { … }
};  // NON si può fare in C++
// Il template indica tante possibili funzioni che non possono essere tutte trattate virtualmente

template <...>
class C {
  virtual void foo();
}  //SI PUÒ fare
// Fissati i valori del template, esiste un solo metodo possibile da trattare come virtual
```

Multy-Threading e Multy-processing

Supporto alla concorrenza (nuovo, che nel C++ 2003 non esiste)
- Abbiamo sempre ragionato su un singolo flusso di istruzioni (un solo thread):
Program counter (esegue istruzioni), heap (memoria), memoria statica, stack, lista istruzioni etc
- Le macchine moderne hanno più circuiti con tante CPU che possono eseguire più cose contemporaneamente. Come si può sfruttare questo eccesso di hardware?
1) Multy-processing: ci sono in esecuzione più processi (istanza di un programma in esecuzione; se ne apro 50, ne avrò 50 uguali che agiscono senza interferire se non lo decido io).
Ogni processo è appunto separato, ma serve un modo per farli interagire.
Per interagire, hanno 2 modalità:
- 1: comunicare attraverso effetti esterni (un processo scrive su un file e un altro processo legge sullo stesso file).
- 2: chiedere al sistema di avere un blocco di memoria condiviso.
Il fatto che siano indipendenti però è una cosa buona perché si evitano problemi (possono essere distribuiti più facilmente).

Per farli interagire invece si fatica e si rallenta (più lenti rispetto la situazione ideale).
2) Multy-threading: ho tante diverse CPU, ognuna con il suo program counter, ognuna con il suo flusso di istruzioni (che spesso provengono dallo stesso programma), ognuna con il suo stack MA hanno in comune la memoria statica e la heap. Nel complesso è come un singolo processo distinto dagli altri, con però dei problemi:
- Potrebbero esserci interferenze.
- Possono condividere dati in modo semplice (e sbagliare più facilmente).
Lato positivo:
- È più probabile che la chache venga usate maggiormente e in modo più efficiente.
(è molto di moda anche se ora la cosa più in moda è il multy-processing (Facebook, Google, etc))

Nel C++ 2003 si poteva fare ma non era standard (si poteva scrivere in modo diverso ed era quindi più difficile); nel 2011 si è inserita una libreria standard.
Ora la concorrenza è diminuita di "una variabile" (prima bisognava decidere quale libreria ora no).
Il problema di gestire tanti processi multy-trading rimane anche con la libreria standard.
Quando bisogna decidere cosa usare, si deve capire com'è il programma che si vuole fare:
- E' ovvio, evidente che potevi fare le cose in parallelo (es: applicazione che deve fare OCR, è ovvio che è meglio processare più pagine contemporaneamente).
Ogni volta che si crea uno stack si occupa 1 MB quindi è meglio non crearne troppi.
Bisogna capire quanti thread la macchina può fare per ridurre gli stack.
Qui l'unico limite è appunto sapere quanti ne posso fare (ma che devo usare i thread è ovvio).
- Se i thread devono interagire spesso (usare uno strumento a turni) in tal caso bisogna vedere quanto questa interazione riduce l'efficienza e qui le cose diventano complicate.
Ogni volta che una istruzione viene usata da più thread, si rischiano errori.
Quando abbiamo parlato di invarianti dicevamo che l'invariante della classe va controllata alla fine di una istruzione e alla fine di ogni metodo che potrebbe aver modificato l'oggetto ma questo vale in modalità single-thread.
Race-condition: thread che fanno gara a chi arriva prima (il programma non è più deterministico ed è normale MA anche quando non ci sono errori, fare testing è difficile perché non so come dovrà essere esattamente l'output... questo non è un errore ma le cose possono andare male).
Leggo il biglietto - ne ho uno – decremento - ottengo 0 biglietti SE sono atomiche;
Ma se io le divido per due utenti diversi che accedono allo stesso sito, uno comprerà il biglietto ma l'altro no perché qui le operazioni sono divise.
Data-race: chi arriva prima, fa una operazione.
La libreria che supporta il multy-thread cerca di regolare questo traffico. Si controlla il data-race per regolare il traffico.
Dead look: entrambe hanno una risorsa ed entrambi aspettano che l'altro liberi la sua risorsa. È un errore grande ma ci si accorge della sua presenza.
Il caso peggiore è quello in cui il sistema non muore ma va avanti con tante risorse in modo lentissimo.
La programmazione concorrente è un problema tosto.
Se un programma multy-thread non funziona è MOLTO difficile correggerlo; facendo testing gli errori non sono più sistematici; il sistema può aiutarti al massimo controllando la consistenza.

Quando usare il multy-thread?
- Voglio migliorare le performance.
- Semplificare la programmazione: in alcuni casi l'applicazione DEVE fare cose contemporaneamente ma queste cose sono indipendenti (es: corpo umano) e farle in un unico thread creerebbe molto disordine.

In questo caso è più facile ragionare con i multi-thread.

Esempio Multy-thread (solo C++ 2011)

- Ricordare di dire al compilatore che vogliamo usare il Multi-thread dicendogli che vogliamo usare la libreria pthread.
- Il thread del main, è separato da quelli dichiarati da noi quindi si rischia di chiudere il programma (main) con ancora dei thread attivi (errore); si risolve con il .join()

```
#include <iostream>
#include <thread>
#include <cvector>

void hello(int i) { //Oggetto funzione - collable
  std::cout << "hello" << i << endl;
}

int main() {
  std::vector<std::thread> vt;

  for(int i=0; i < 10; i++) {
    vt.push_back(std::thread(hello,i);
  } //Stampando si vede che non vengono stampate in ordine se si usa printf al posto di cout (nella funzione hello); se si usa cout (nella funzione hello) si ha una “corsa” nel volerlo utilizzare e quindi l'output è ancora più disordinato, anche perché si hanno 3 funzioni, una per ogni << che è appunto l'operatore che chiama una funzione e il cout quindi non è atomico, printf si.

std::thread t(hello, 1); //Altra funzione nuova che permette di scrivere così come se fosse scritto std::thread t(hello(1);)

t.join(); //Si riunisce al thread principale

return 0;
}
```

- Con il multy-thread aumenta il problema dei dangling pointer
- Di default si passano copie dei dati e non quelli originali.
- Idealmente dividendo il processo in 10 parti, si aumenta di 10 volte la velocità ma nei thread non funziona perché in realtà esiste sempre una parte del codice sequenziale (non divisibile).

Con la legge di Amdahl si calcola quanto aumenta di velocità usando i thread con la formula:

$1 / ((1-P) + P/s)$

- A livelli alti, si pensa di usare il C++ perché da la possibilità di controllare il costo di quello che state facendo (anche nei videogiochi, rendering di C4D, etc si usa il C++).

//FINE

ESERCIZIO 3 (mostrare output) - 22/2/2005

```
#include <iostream>
class C1 {
public:
   C1() {
      std::cerr << "Constructor C1::C1()" << std::endl;
   }
   ~C1() {
      std::cerr << "Destructor C1::~C1()" << std::endl;
   }
};
class C2 {
public:
   C2() {
      std::cerr << "Constructor C2::C2()" << std::endl;
      throw 123;  //lanciando errore, esce sempre
   }
   ~C2() {
      std::cerr << "Destructor C2::~C2()" << std::endl;
   }
};
class C3 {
public:
   C3() {
      std::cerr << "Constructor C3::C3()" << std::endl;
   }
   ~C3() {
      std::cerr << "Destructor C3::~C3()" << std::endl;
   }
};
class D {
private:
   C1 c1;
   C3 c3_1;
   C2 c2;
   C3 c3_2;

public:
   D() : c1(), c2(), c3_1(), c3_2() {  // ”c3_1” VIENE PRIMA di “c2” perché vale l’ordine di come sono dichiarati nel private.
      std::cerr << "Constructor D::D()" << std::endl;
   } // I sottoggetti che danno errore non ci sono; verranno eliminati solo gli oggetti che sono stati creati; se ho errori il D non è stato creato.
   ~D() {
      std::cerr << "Destructor D::~D()" << std::endl;
   }
};
```

```
int main() {
   try { // Si controlla che tutto avvenga nel modo giusto (ma si genera errore nella creazione di c2; d non viene creato)
      D d;
   }
   catch (char c) { // Se esiste un modo per convertire un intero in un carattere? No, quindi si esegue il catch successivo; comunque le eccezioni vanno lanciate sempre per valore e non per puntatore perché l'oggetto puntato potrebbe diventare dangling. Inoltre le eccezioni non vanno mai catturare per valore ma per riferimento (costante); qui non sono state catturate nel modo migliore.
      std::cerr << "char " << c << std::endl;
   }
   catch (...) {
      std::cerr << "..." << std::endl;
   }
return 0;
}
```

Il catch non fa conversioni standard ma fa quelle da classe base a derivata (comunque meglio usare & e non ci sono problemi).
Se si lancia una eccezione, e non si cattura succedono varie cose dipende dalla macchina.
- Nelle eccezioni non c'è overloading; il primo catch che funziona viene usato nascondendo gli altri.
Catturare un oggetto derivato da una classe T, cattura anche le eccezioni di tipo T.

Soluzione (output):
Constructor C1::C1()
Constructor C3::C3()
Constructor C2::C2()
Destructor C3::~C3()
Destructor C1::~C1()
...

ESERCIZIO 4 - 6/2/2006

Il codice seguente aggiunge al secondo argomento una copia degli elementi di tutte le liste contenute nel primo argomento (una lista di liste), realizzando quindi una sorta di concatenazione multipla.

```
#include <list>
using namespace std;
template <typename T>
void f(const list<list<T> >& ll, list<T>& l) {  // la prima è costante
    for (typename list<list<T> >::const_iterator ll_i = ll.begin(),
        ll_end = ll.end(); ll_i != ll_end; ++ll_i)
    for (typename list<T>::const_iterator i = ll_i->begin(),
        i_end = ll_i->end(); i != i_end; ++i)
    l.push_back(*i);  // l è la lista finale che contiene le altre
}
```

Generalizzare il codice affinché, mantenendo la funzionalità descritta, possa lavorare non solo con il tipo lista, ma con combinazioni qualunque dei contenitori sequenziali della STL. Generalizzare ulteriormente il codice affinché accetti come argomenti, invece di contenitori sequenziali della STL, degli iteratori. Fornire un (semplice) esempio di applicazione della versione con iteratori che non possa essere codificato utilizzando la versione con contenitori.

Osservazioni:

- E' una lista di liste.
- Concatena le liste per crearne una.
- Deve funzionare con vector, stack, liste quindi generalizzare il codice usando i contenitori.
- Dobbiamo usare un contenitore qualunque (il quale può essere uno dei tipi precedenti)..

Soluzione con contenitori:

```
 using namespace std;

 template <typename Cont1_di_cont2, typename Cont3> // Vogliamo parametrizzare per il nostro contenitore; ho un contenitore di tipo 1 che contiene altri contenitori di tipo 2
 void f2(const Cont1_di_cont2& c1c2, Cont3& c3) { // Il c3 è il contenitore di uscita che può essere diverso da quelli di entrata

 typedef  typename Cont1_di_cont2::value_type Cont2; //value_type qui indica il tipo di ciò che è contenuto nel Cont1_di_cont2; typename gli dice che sto tirando fuori un nome di tipo e non un tipo di variabile

 for (typename Cont1_di_cont2::const_iterator c1c2_i = c1c2.begin(),
      c1c2_end = c1c2.end(); c1c2_i != c1c2_end; ++c1c2_i) //Ora dobbiamo iterare sugli oggetti di Cont1_di_cont2 e ci servirebbe sapere il Cont2 (tipo del contenitore interno); definiamo questo tipo fuori dal for (o anche dentro) con il typedef
   {  for (typename Cont2::const_iterator i = c1c2_i->begin(),
          i_end = c1c2_i->end(); i != i_end; ++i)
      c3.push_back(*i);
   }
 } // Fine f2
```

- Ho usato c3 perché nessuno ci obbliga ad avere come risultato qualcosa di tipo cont1 o cont2
- Prima il codice dipendeva dalle liste, mentre quello fatto da noi non necessita della stl (ovviamente quando lo dobbiamo utilizzare però dovremmo definire dei contenitori)
- DA NON FARE:

```
template <typename Cont1< Cont2<T> > >
template <typename Cont1_di_cont2 <T> >
```

Soluzione con iteratori:

```
 using namespace std;

 template <typename Iter1_di_cont2, typename Out_Iter3>
 Out_Iter3 f3(Iter1_di_cont2 first, Iter1_di_cont2 last, Iter3 out) {

 typedef typename std::iterator_traits<Cont_di_cont2>::value_type Cont2; //Gli iteratori puntano ad oggetti di tipo cont2; funziona se ho un iteratore furbo che può sapere il value_type; per sicurezza uso un tutore che è iterator_traits

   for ( ; first != last; ++first) //Più semplice del precedente
   {
    for (typename Cont2::const_iterator i = first->begin(),
          i_end = first -> end(); i != i_end; ++i , ++out )
      *out = *i; //Scrivere in una sequenza di out
      // ++out;
    }
  return out;
}
```

- Possiamo usare cose che parzialmente non sono contenitori (quindi più generico).
- La sequenza più esterna (che contiene le altre) potrebbe non essere un contenitore, la stessa cosa per l'output (non è un contenitore quindi posso scrivere anche direttamente su cout).
- Con la push_back si fa una copia (creazione); negli iteratori si può lavorare in modalità sovrascrittura e quindi più efficiente (per defaul si sovrascrive).
- Questi codici funzionano solo per quei tipi che definiscono le funzioni usate al loro interno.

Perché in questo esercizio abbiamo usato i typename?
Esempio: mi vedo arrivare A * B ma non so cos'è * . Non si sa se è una somma come non si sa il tipo di A e B; nel dubbio però il compilatore fa una scelta e di default sceglie che A e B sono nomi di un valore.
Se il compilatore si vede arrivare un const_iterator lo considera di default come valore ma a noi serve come tipo e quindi specifico davanti che è un tipo (typename).

Teoria:
- Contenitori: conteiner delle stl; oggetti nei quali i miei valori sono memorizzati fisicamente.
- Sequenza: è un concetto più astratto (non fisico, non concreto) che indica qualsiasi cosa che si comporta come sequenza; un contenitore sequenziale ad es, è una sequenza (coppia di iteratori di inizio e fine).
- Iteratore: come concetto astratto (è un iteratore tutto ciò che si comporta come definito

dall'iteratore).
Anche i puntatori di una stringa possono essere iteratori.
Io stabilisco cosa mi serve (es: iteratore di input, mi serve che possa copiare, confrontare, incrementare, leggere; se lo fa è iteratore di input altrimenti no).

Extra:
- polimorfismo statico: non mi interessa da dove vieni, basta che tu sappia fare certe cose.
- polimorfismo dinamico: se sono un puntatore alla classe animale, devo fare solo certe funzioni e mi arriva uno che ne fa una in più, allora non va bene (non c'è il datatype_sistem, nell'altro si).

ESERCIZIO 6 (gestione memoria) - 6/2/2006
Discutere brevemente se, e sotto quali condizioni, le seguenti classi si comportano correttamente dal punto di vista dell'allocazione delle risorse (si noti che quella mostrata è da considerarsi l'implementazione completa delle classi in questione).

```
#include <vector>
template <typename T>
class A {
private:
   int i;
   T* p;
   double d;
public:
   A() : i(), p(new T), d() {}
   ~A() { delete p; };
};
// A: definire private la funzione copia e assegnamento (come nella classe B successiva) per evitare che di default facciano cose inaspettate

template <typename T>
class B {
private:
   int i;
   T* p;
   double d;
   B(const B&);  // Copia
   B& operator=(const B&);  // Assegnamento
public:
   B() : i(100), p(new T[i]), d() {}  // In questa dichiarazione delle variabili, esse devono essere dichiarate in ordine (PRIMA la i altrimenti il vettore è di dimensione indefinita); questo è già stato fatto quindi la classe B va bene
   ~B() { delete[] p; };  // p deve essere cancellata come array e così è stato fatto
};
```

```
class C : private A<std::vector<int> > {
private:
   B<double> b1;
   B<int> b2;
   C(const C&);
   C& operator=(const C&);
public:
   C() {}
};
// Se C usa assegnamento e copia di A (funzioni che in A sono problematiche come visto in precedenza) allora non va bene.
Se invece non le usa, allora C è giusta (anche se in A non è stata fatta la "correzione" indicata all'inizio; l'importante è non usare le due funzioni assegnamento e copia).
```

Soluzione nei commenti.

Esempio typedependent:

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

//foo();

template <typename T>
struct S : public T { //Si può fare
  void bar() {
    foo(); //this->foo();
  }
}
```

Dandola al Gcc esso restituisce errore (il vecchio Gcc non lo faceva); cos'è questo errore?
- Noi stiamo compilando il template S e vogliamo vedere che lui si comporti bene (la chiamata di foo per il compilatore non dipende da T anche se per noi in realtà ne dipenderà in futuro).
- Il compilatore (arrivato alla foo di bar) cerca la foo ma non la trova e quindi questo è un errore per il compilatore.
- Se avessi la foo fuori (quella commentata) allora il compilatore crea un collegamento permanente a quella foo e non a quella di Base quindi non vi è ereditarietà dalla foo di base.
- Nelle classi templatiche che ereditano, per essere tranquilli, è meglio scrivere this->foo(); così ora dipende da T e quindi il compilatore compilerà se e solo se nella classe Base da cui eredita c'è foo().

C++ 2011/2014 – Mondo lavorativo (extra)

Nella realtà lavorativa attuale, si usa il C++ 2003 ma esiste già la versione 2011/2014.
- Lato negativi: il linguaggio che ha successo deve mantenere la compatibilità con quello vecchio e questo non va bene.
- Lato positivi: se devono inserire nuove caratteristiche ci vanno molto piano, non le fanno per moda, e quindi funzionano meglio e sono più utili (sono passati 11 anni per il C++ 2014)
- Chiedersi sempre perché qualcosa viene introdotto per capire come usarlo.

Se non sapete PERCHÉ esiste un costrutto, allora:
- Quel costrutto per voi non esisterà.
- Non ci si rende conto che è necessario anche se invece servirebbe (in passato avevo un problema; posso risolverlo con il nuovo codice?)

Introduzioni del C++ 2011

Enumeratori:

Tipi enumerazione: enum si può usare per elencare un certo numero di costanti per nome.
È stato esteso nello standard nuovo.

```
enum Colors {
RED, //0 – se non si dichiarano valori assegna in automatico i valori da 0 in poi
GREEN, //1
BLUE //2
};
enum Glof_Place {
TEE,
FAIRWAY,
RUGHT,
BUNKER,
GREEN
};
```

1) enum definisce un tipo ma non uno scope: GREEN è stato definito due volte per ambiti diversi quindi esistono due GREEN che vanno in conflitto perché si vedono anche fuori dalle { } (fuori dallo scope).
2) Ogni compilatore scegli quale tipo associare all'enumerazione (in genere intero); per poter determinare quanto spazio occupare, il compilatore deve aver visto tutte le variabili (non posso fare un enum templatico).
- Nel C++ 2003 i tipi enum NON possono essere dichiarati senza dire come sono fatti; vanno sempre dichiarati e definiti.

Problemi di ciò: se si fa un enum per codificare gli errori, allora bisogna mettere tutti i codici di errore nell'header file (se cambio la versione aggiungendo o togliendo un codice di errore, tutti gli utenti che usano la vostra libreria devono ricompilare tutti i file dato che è cambiato l'header file).

- Nel C++ 2011 si introducono le Enumerazioni con scope:

```
enum class Colors {
RED = 3,
GREEN, //4
BLUE //5
};
```

```
enum class Glof_Place {
TEE,
FAIRWAY,
RUGHT,
BUNKER,
GREEN
};
```

1) GREEN non si vede fuori dalla classe quindi non fanno interferenza.
Per usare i GREEN fuori, dovrò scrivere:
Golf_Place gp = Golf_Place::GREEN;
2) Nel caso dei codici di errori, come faccio ad aggiungerli senza dover ricompilare tutto?
Prima non potevo perché il compilatore non sapeva quanto spazio di memoria allocare e doveva conoscere tutti gli errori quindi ora si da la possibilità al linguaggio di indicare lo spazio che occuperà (scegliendo il tipo):

```
enum class Colors : unsigned char ;

enum class Golf_Place : int;
```

3) Problema: non sono fortemente tipate; si possono convertire automaticamente da enum a int (ma non accetta il contrario, probabilmente) e questo può causare problemi quindi meglio non fare conversioni automatiche (se si usa class).

Nel C++ VECCHIO c'era una soluzione al problema 1):
1) Scope: si risolve con i namespace

```
namespace Enum_Color {
  enum Color {
  RED = 3,
  GREEN, //4
  BLUE, //5
  };
}

Enum_Color::Color gp = Enum_Color::GREEN;
```

&& R-Value

```
struct T {
S();

S(const S& other) = default; //Copia costruttore
S& operator = (const S& other) = default; //Copia assegnamento

S(S&& other) = default; //Muovi il costruttore
S& operator=(S&& other) = default; //Muovi l'assegnamento
//Prende le risorse e le mette dentro l'oggetto che sto usando;si passa un L-value che sta per morire e
```

```
il quale da tutti i suoi valori a qualcun altro senza copiarli e perdendo la “scatola” precedente.
//E’ una ottimizzazione che non importa a chi programma a basso livello

~S() = default;

};

struct T { //Fa la stessa cosa di quella sopra ma è meno specifica
  //Vuota
};
```

- Se si vuole essere specifici, si hanno 3 motivi:
Didattica: per insegnare a qualcuno.
Documentazione: far conoscere all'utente come funziona.
Insicuro: per sicurezza, per sapere cosa succede.

Smart Pointer

```
struct T {
  R* risorsa;

  T() : risorsa(new R()) {}

  ~T() { delete risorsa;}

  T(const T& t) risorsa (new R(*t.risorsa)) {}

  //Assegnamento...
};
```

- Qui vi era il problema risolvibile con classe RAII ma lo abbiamo risolto in un altro modo perché esistono altri modi:

```
#include <memory> //Applica in modo intelligente l'ideoma RAII

struct T {
  std::unique_ptr<R>R risorsa; //Unique: sono l'UNICO che può usarla, si distrugge con me.
  std::share_prt<R> risorsa; //Share: possono lavorarci in molti e la risorsa si elimina quando l'ultimo smette di usarla; si alloca sia la risorsa che i dati necessari per sapere quanti la stanno usando.
  T() : risorsa(new R()) {…} //Va bene

//Gli altri due seguenti si possono non scrivere perché il rilascio viene gestito da unique_ptr; anche il move viene gestito da ptr.
  ~T() { delete risorsa; }
  T(const T& t) risorsa (new R(*t.risorsa)) {…}
  //La copia non posso farla con l’Unique perché non lo permette; si può muovere ma non copiare.
};
```

Usando i puntatori smart si evita di scrivere "molto" codice.

- Per sapere quale puntatore smart usare, devo sapere COME voglio usare la risorsa:
) Se la nostra funzione (è un pozzo e) si prende la risorsa, allora e meglio che sia passata con un puntatore furbo (smart).
) Altrimenti la passo con i classici puntatori.
Es) La fattoria costruisce l'animale e restituisce un puntatore nudo; in quel caso si può usare un puntatore smart il quale muore quando muore la risorsa.

Weakptr (altro tipo): la risorsa può essere eliminata anche se il puntatore non muore.
- Il puntatore nudo non ha modo di sapere se ciò che punta esiste ancora o no.
- Il weakptr può invece chiedere se la risorsa c'è ancora; se c'è la si aggancia e si crea uno Share_ptr
- Permette una gestione più sicura evitando memory leek o dangling pointer.

Esercizi d'esame

ES - 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 N {
   struct C {
      C(int);              // funzione #1
      C(const C&);         // funzione #2
   };

   void f(double d);       // funzione #3
   void f(const C& c);     // funzione #4

   void g(int i, double d); // funzione #5
   void g(int i, int j);    // funzione #6

   void h(C* pc);          // funzione #7
} // namespace N
void f(char);              // funzione #8

int h(const char* s = 0);  // funzione #9
int h(const N::C* pc);     // funzione #10

int main() {
   N::C c(5);    // chiamata A
   f(5);         // chiamata B
   f(c);         // chiamata C
   N::f('a');    // chiamata D
   g(5, 3.7);    // chiamata E
   N::g(2.3, 5); // chiamata F
   N::g(5, 2.3); // chiamata G
   h(&c);        // chiamata H
   ...
}
```

	Candidate	Utilizzabili	Migliore
A	1 , 2	1 , 2	1
B	8	8	8
C	8 , 3 , 4	4	4
D	3 , 4	3 , 4	3
E	/	/	/
F	5 , 6	5 , 6	6
G	5 , 6	5 , 6	5
H	7 , 9 , 10	7 , 10	7

- A: 2 è utilizzabile tramite il costruttore (Funz. 1) che usando il 5 crea un oggetto di tipo C.
- D: non si considerano le funzioni globali; 3 è utilizzabile perché esiste conversione da char a double; 4 è utilizzabile grazie al costruttore definito dall'utente che usa 'a' per costruire un oggetto di tipo C.
- E: non ci sono candidate.
- H: 9 non è utilizzabile perché non esiste conversione da puntatore a valore costante.

ES - 2) Scrivere un esempio, minimale ma completo, di codice che causa un errore di compilazione a causa dell'assenza delle guardie contro l'inclusione ripetuta. Spiegare brevemente il motivo dell'errore.

```
//file 1.hh
struct S {};
...

//file 2.hh
#include "File 1.hh"
...

//file.cc
#include "File 1.hh"
#include "File 2.hh"
...
```

L'errore si può correggere gggiungendo le guardie in "file 1.hh" e "file 2.hh".
L'errore è la doppia definizione di S (violazione ODR).

ES - 3) Si supponga che il metodo booleano check_inv() restituisca il valore true se e solo se l'oggetto sul quale `e invocato soddisfa la proprietà invariante della classe C. Indicare, nel codice seguente, come ed in quali punti è opportuno inserire il controllo di tale invariante allo scopo di semplificare le attività di testing e debugging della classe (si assuma che il codice non contenga nessuna occorrenza di const_cast).

```
class C {
public:
   bool check_inv() const {
      // Codice che implementa il controllo di invariante.
   }

   C(int a, int b) {
      // Implementazione del costruttore.
   }

   void foo(C& y) {
      // Codice che implementa il metodo foo.
   }

   void bar(const C& y) {
      // Codice che implementa il metodo bar.
   }

   void ying(const C& y) {
      // Codice che implementa il metodo ying.
   }

   void yang(const C& y) const {
      // Codice che implementa il metodo yang.
   }

   ~C() {
      // Codice che implementa il distruttore.
   }

   static void zen(int i, double d) {
      // Codice che implementa il metodo zen.
   }

   // ... altro codice ...
}; // class C
```

Uso gli assert per trovare l'errore.
Utilizzo dei controlli:
- Alla fine del costruttore.
- In foo: si fa il controllo per entrambi i membri alla fine della funzione (per il this e y; per il this

posso scrivere "this.check_inv()" o "check_inv()" dentro ad assert).
Posso fare anche i controlli all'inizio (se li faccio in questa prima funzione, li faccio in tutte).
- In bar: alla fine controllo solo il this perchè y è const.
- In ying: stessa cosa di bar.
- In yang: nessun controllo (o controllo solo quello in ingresso).
- Nel distruttore: controllo solo eventualmente l'oggetto all'inizio prima di distruggerlo.
- In zen: è static (cioè, è come se non avesse il this) e i parametri non sono di tipo C quindi non c'è niente da controllare.

ES - 4) Si forniscano il prototipo e l'implementazione della funzione generica transform i cui parametri specificano: (a) due sequenze di ingresso (non necessariamente dello stesso tipo); (b) una sequenza di uscita (potenzialmente di un tipo ancora diverso); (c) una funzione binaria applicabile a coppie di elementi in ingresso (il primo elemento preso dalla prima sequenza, il secondo elemento preso dalla seconda sequenza) e che restituisce un elemento assegnabile alla sequenza di uscita. La funzione transform applica la funzione binaria agli elementi in posizione corrispondente nelle due sequenze di ingresso, assegnando il risultato ad elementi consecutivi della sequenza di uscita, ed arrestandosi quando si raggiunge la fine di una qualunque delle due sequenze di ingresso (Nota Bene: le due sequenze in ingresso hanno lunghezze arbitrarie). Utilizzando la funzione suddetta, scrivere una funzione che, date due liste di numeri floating point [a1,...,an] e [b1,...,bn], aggiunge in coda ad una terza lista le medie aritmetiche.

Le sequenze usano iteratori e non contenitori.
Gli algoritmi stl di solito usano iteratori (perchè non vogliono avere a che fare con i contenitori).

```
template <typename Iter1, typename Iter2, typename Out, typename BinaryFunction>
Out   // Restituisco Out per poter concatenare
transforms(Iter1 first1, Iter1 last1, Iter2 first2, Iter2 last2, Out out, BinaryFunction fun);
// La sequenza di output ha un unico parametro perchè non conosco la fine.
// La funzione non è un predicato perchè può restituire qualcosa che non è bolo.
{
  for( ; first1 != last1 && first2 != last2; ) {
    *out = fun(*first1, *first2);
    ++first1;  ++first2;  ++out;
  }
  return out;
}

float media(float a, float b) {
  return (a+b)/2;
}

void foo(const std::list<float>& lista1, const std::list<float>& lista2, std::list<float>& lista3) {
  transform(lista1.begin(), lista1.end(), lista2.begin(), lista2.end(), std::back_insert(lista3), media);
// Uso il contenitore "std::back_insert(…)" perchè "lista3.end()" andrebbe a scrivere dove non si può invece con insert si può aggiungere anche in fondo un elemento grazie ad un riallocamento che l'insert fa in automatico in caso serva spazio
}
```

ES - 5) Il codice seguente non si comporta bene, dal punto di vista della gestione delle risorse, in presenza di eccezioni. Individuare almeno un problema, indicando la sequenza di operazioni che porta alla sua occorrenza. Fornire quindi due soluzioni alternative:
• la prima basata sull'utilizzo dei blocchi try . . . catch;
• la seconda senza utilizzare alcun blocco try . . . catch.
Per entrambe le soluzioni si richiede che l'ordine di allocazione e deallocazione delle risorse di tipo A sia lo stesso codificato nel codice originale (per il caso di esecuzione senza eccezioni).

```
void foo() {
   A* a1 = new A(1);
   A* a2 = new A(2);
   job1(a1, a2)
   delete a2;
   A* a3 = new A(3);
   job2(a1, a3)
   job3(a3)
   delete a3;
   delete a1;
}
```

---

1) Con il try-catch:

```
void foo() {
   A* a1 = new A(1);
   try {
     A* a2 = new A(2);
     try {
       job1(a1, a2)
       delete a2;
     }
     catch (…) {
       delete a1;
       throw;
     }
     A* a3 = new A(3);
     try {
       job2(a1, a3)
       job3(a3)
       delete a3;
     }
     catch (…) {
       delete a1;
       throw;
     }
     delete a1;
   }
   catch (…) {
     delete a1;
     throw;
   }
}
```

2) Classe RAII:

```
struct RA {
private:
  A* a;
  RA(const RA&);
  void operator=(const RA&);

public:
  RA(int i) : a(new A(i)) {
  }

  RA() {
    delete a;
  }
  A* get() { return a; }
};

void foo() {
   RA a1(1);
   {
      RA a2(2);
      job1(a1.get(),a2.get());
   }
   RA a3(3);
   job2(a1.get(),a3.get());
   job3(a3.get());
}
```

2) Usando la classe "A" già esistente:

```
void foo() {
   A a1(1);
   {
     A a2(2);
     job1(&a1, &a2);
   }
   A a3(3);
   job2(&a1,&a3);
   job3(&a3);
}
```