**Canto zero: dal nulla all'esecuzione del PRIMO programma**

**int main(   ){   }**

Questo è il programma C++ più breve che possa essere scritto senza che il compilatore segnali alcunché di errato, e che può essere formalmente eseguito con successo quantunque non faccia assolutamente nulla. Come tale presenta quanto ci deve essere di irrinunciabile in un normale documento di testo per poter affermare che si tratta di un programma scritto in C++, vale a dire:

* una funzione chiamata obbligatoriamente *main* cui si attribuisce il tipo *int* che ne precede il nome;
* una lista di argomenti per tale funzione, che ne segue il nome, racchiusa tra parentesi tonde e inserita obbligatoriamente anche se vuota come nel caso presente;
* un cosiddetto ambito della funzione, racchiuso tra parentesi graffe e inserito obbligatoriamente anche se vuoto come nel caso presente.

L'unica spaziatura necessaria è quella che separi la parola *int* dalla parola *main*; oltre a questa ne possono essere inserite quante altre se ne vogliano in qualsiasi posizione purché non si spezzino le due uniche parole presenti nel testo, che devono essere considerate a tutti gli effetti come gli atomi di Democrito. In altri termini, se il testo presente si riscrive così:  
  
int        main  
  
(  
  
                 ){  
  
  
  
}   
  
al compilatore va altrettanto bene, anche se vien da chiedersi quanto abbia bevuto l'autore del codice, mentre non verrebbe accettato se fosse scritto:  
  
int ma    in(  ){  }  
  
a causa della spaziatura illecita che spezza la parola *main*.  
Il programma nullafacente fin qui introdotto va scritto, ovviamente, in un documento di testo usando esclusivamente un normalissimo *text editor*, **assolutamente NON** un *word processor*. Per intendersi si rifugga dall'idea di scrivere codici C++ usando programmi comunemente denominati *suites per ufficio* (*word*, *wordpad*, *office* et similia) dato che questi introducono nel documento pletore di *bytes* di formattazione che il compilatore rifiuterebbe come la peste, e lo fanno alla piena insaputa dell'autore del codice.  
Un *text editor* (*gedit*, *emacs*, *kate* et similia) invece, introduce nel documento solo ed esclusivamente i caratteri che la mente del programmatore, guidando i polpastrelli delle sue stesse dita sulla tastiera, intende effettivamente e volontariamente che siano inseriti.  
  
Molti *text editors*, per giunta, non sono affatto stupidi e sono in grado di riconoscere le parole appartenenti al vocabolario del linguaggio e molti dei suoi costrutti sintattici, evidenziandoli con colori convenzionali che sono di considerevole aiuto durante la redazione del codice.  
Il nome del documento che contiene il testo del programma è a totale discrezione dell'autore, ma è molto opportuno che termini con una delle sequenze di caratteri preferite dal compilatore che sono:  
  
**.c**      **.C**        **.cc**      **.cpp**        **.CPP**      **.c++**        **.cp**      **.cxx**         
  
Se qualcuno/a volesse fare l'originale e battezzare il proprio documento contenente il programma con lo stesso nome della/del sua/o ragazza/o, si vada a cercare nelle 181 pagine della documentazione del compilatore quale opzione deve fornirgli affinché sia disposto a ignorare il fatto che il nome del documento ha un *suffisso* sconosciuto.  
  
Siccome io non ho tempo da perdere, e non ho neppure la ragazza, supporrò senz'altro che il nome del documento contenente il programma sia   
  
            pippo.C   
  
e, per compilarlo, ossia per tradurlo in codice binario eseguibile dal processore, sarà sufficiente dare alla *shell* il comando  
  
             g++    -std=c++11   pippo.C  
  
La *shell* risponderà ripresentando semplicemente il proprio *prompt*, ossia disponendosi a ricevere il comando successivo, beninteso se il compilatore C++ è correttamente installato, altrimenti risponderebbe *Comando non trovato* o altre amenità.  
  
L'opzione *-std=c++11* attiva l'interpretazione del codice nello *standard* 2011 del linguaggio: è disponibile sul compilatore GNU di Linux dalla versione 4.7; se qualcuno ha una versione precedente si suggerisce o un aggiornamento del sistema, oppure NON USARE costrutti sintattici propri di quello *standard*. La prima opzione è caldamente raccomandata.  
  
Per evitare di dovere scrivere sempre quell'opzione si potrebbe creare un cosiddetto *alias* del comando che la contempli implicitamente, ad esempio inserendo nel documento di configurazione della propria *shell* una linea del tipo  
  
*alias g++='/usr/bin/g++ -std=c++11'*  
  
Frattanto, in un modo o nell'altro, il compilatore avrà creato nella cartella corrente un nuovo documento dal nome criptico  
  
            a.out  
  
che contiene appunto il codice eseguibile e può essere mandato in esecuzione digitando             ./a.out  
  
terminato con Invio: il TASTO *Invio*, NON la parola I-n-v-i-o, a destra del *prompt* della *shell*: non accadrà nulla, se non la reiterazione del *prompt*.  
Evidentemente per ottenere qualcosa occorre inserire delle istruzioni nel programma e queste vanno poste all'interno dell'ambito di *main*, ossia entro le graffe; ad esempio, per far apparire sul terminale il risultato di 2+2, si potrebbe scrivere  
  
**#include <iostream>  
int main(   )  
{std   ::    cout << "2+2 fa " << 2+2 << '\n';}**  
  
Come si vede, la semplice introduzione di un'istruzione, meglio dire: un'*espressione*, che produca un risultato così banale ha provocato la comparsa sulla scena di parecchi nuovi personaggi:

* un *oggetto* chiamato **std   ::   cout**, il cui nome NON fa parte del vocabolario del linguaggio e per la cui comprensione da parte del compilatore è NECESSARIO l'inserimento della misteriosa prima riga, che precede *main*; questo oggetto si incarica di inserire dati nel cosiddetto *standard output* del programma, ossia, salvo avvisi contrari, sul terminale da cui il programma stesso verrà eseguito: va da sé che qualsiasi programma che faccia qualcosa di utile non potrà mai esimersi dall'usare oggetti di questo genere.
* un *operatore* (il segno <<, senza alcuno spazio tra i due <) che dice a **cout** che cosa inserire, e come, nello *standard output*: ciò che lo segue immediatamente.
* una *costante stringa* ("2+2 fa ") che, trovandosi a destra dell'operatore, sarà inserita tale e quale nello *standard output*;
* una *sottoespressione* aritmetica (2+2) che come tale sarà valutata e il cui risultato, trovandosi anch'essa a destra di <<, sarà pure inserito in *output*;
* una *costante carattere* ('\n'), pure inserita in *output*, e che causerà una semplice *andata a capo*, utile per la leggibilità dell'*output* stesso.
* il segno finale di punto e virgola OBBLIGATORIO perché occorre comunque segnalare al compilatore la fine dell'espressione, ANCHE quando ce ne fosse una sola come nel caso presente.

Compilando novamente *pippo.C* con l'attuale contenuto e rieseguendo *./a.out* (FATELO!) si vedrà apparire sul terminale la confortante affermazione  
  
2+2 fa 4  
  
seguita, a capo, dal *prompt* della *shell*.  
  
Naturalmente il linguaggio consente applicazioni migliori di quella appena riportata: tanto per procedere gradualmente si potrebbe fare in modo che il programma sia capace di eseguire qualsiasi addizione i cui due addendi siano, in qualche modo, forniti dall'esterno all'atto dell'esecuzione. Questo, per inciso, è insito nel concetto stesso di programmazione: un programma non dovrebbe mai conoscere a priori il valore numerico dei propri dati. Un modo per farlo è il seguente:  
  
*#include <iostream>  
using namespace std;  
  
int main(   )  
{double a, b;  
cout << "fornisci i due addendi ", cin >> a >> b,  
cout << a << '+' << b << " fa " << a+b << '\n';}*Si osservi che, rispetto alla versione precedente, non compare più alcuna costante numerica ma solo costanti carattere e costanti stringa; si sono invece introdotte due variabili (*a*, *b*) che sono state dichiarate appartenenti al tipo *double* immediatamente dopo l'apertura della parentesi graffa. La virgola che separa *a* da *b* e il punto e virgola conclusivo sono obbligatori in una dichiarazione.  
I valori numerici delle due quantità sono richiesti direttamente all'utente all'atto dell'esecuzione del programma e sono introdotti nella sua memoria da parte dell'oggetto **cin** che svolge una funzione speculare rispetto a quella dell'oggetto **cout**, vale a dire che estrae dati, attraverso l'operatore >>, dal cosiddetto *standard input* che, salvo avvisi contrari, coincide con la tastiera del calcolatore.  
Quando l'esecuzione del programma giunge a elaborare la *sottoespressione  
cin >> a >> b,*l'esecuzione stessa si arresta in attesa che qualcuno digiti sulla tastiera qualcosa di riconoscibile come due numeri separati da uno spazio e terminati da Invio. I termini *qualcosa di riconoscibile* sono voluti: significano che se si digita del ciarpame il programma, così com'è scritto, potrebbe arrestarsi o concludersi con esiti fantasiosi.  
Si osservi anche la seconda riga qui introdotta (*using namespace std;*) che consente di riferirsi all'oggetto **cout**, e anche all'oggetto **cin**, senza più dovere, ogni volta, citare anche il cosiddetto *namespace* (**std ::**) in cui tali oggetti sono stati descritti al compilatore, perché tale *namespace* si conviene ora che sia sottinteso come, in pratica, recita la seconda riga. Ciò è estremamente conveniente, specialmente quando, come accade sostanzialmente sempre, gli oggetti **cout**, **cin** e molti altri debbano essere usati numerose volte: ecco il motivo per cui, di fatto, QUASI TUTTI i programmi C++ iniziano con le due righe qui riportate.  
Ricompilando e rieseguendo il programma, come sopra spiegato, si vedrà apparire sul terminale la frase:  
  
*fornisci i due addendi*   
  
con alla destra il cursore lampeggiante: il programma, a quel punto, sta valutando la sottoespressione:  
  
*cin >> a >> b,*e quindi, per poterne venire a capo con successo, occorre appunto fornirgli due numeri, come gli è stato opportunamente detto di richiedere e come già è stato spiegato.  
Se si ottempera alla richiesta comparirà, immediatamente dopo la pressione di Invio, il corretto risultato dell'addizione; se invece si digiterà *la vispa teresa*... provate e vedrete.   
  
**NOTA BENE**: se si vogliono fornire valori non interi la virgola deve essere in realtà un punto; in altre parole il valore approssimato a due cifre decimali di pi greco va dato come 3.14 **NON** come 3,14.   
  
È del tutto evidente che le altre operazioni aritmetiche elementari saranno trattabili allo stesso modo utilizzando gli opportuni operatori, dotati già delle corrette regole di precedenza. La doverosa cautela contro l'eventualità delle divisioni per zero è a carico dell'autore del codice: se si provasse a sostituire, nel programma precedente, l'addizione con la divisione e si digitasse apposta, per la variabile *b*, il valore 0, si avrebbe come risultato *inf*, abbreviativo di infinito, se il dividendo fosse non nullo e *nan*, acronimo di *Not A Number*, se anche il dividendo fosse nullo.  
  
Per concludere quest'introduzione si farà osservare come, in generale, un programma C++ possa essere concepito come un insieme di funzioni che sono eseguite in un certo ordine stabilito dal programmatore, la prima funzione eseguita essendo sempre quella chiamata *main*, eventualmente, come si è fin qui visto, unica. Come esempio si tratteranno appunto le quattro operazioni artitmetiche elementari, la cui valutazione, piuttosto che effettuata come prima in *main*, sarà demandata a quattro distinte funzioni dal nome parlante. Ecco il codice (compilatelo ed ESEGUITELO):  
  
*#include <iostream>  
#include <cstdlib>  
using namespace std;  
  
double quoziente(double dividendo, double divisore)  
{return dividendo / divisore;}  
  
double prodotto(double moltiplicando, double moltiplicatore)  
{return moltiplicando \* moltiplicatore;}  
  
double differenza(double minuendo, double sottraendo)  
{return minuendo - sottraendo;}  
  
double somma(double addendo\_uno, double addendo\_due)  
{return addendo\_uno + addendo\_due;}  
  
int main(int narg, const char \* \* args, const char \*\* env)  
{double a = atof(args[1]), b = atof(args[2]);  
cout   
<< a << '+' << b << " fa " << somma(a, b) << '\n'  
<< a << '-' << b << " fa " << differenza(a, b) << '\n'  
<< a << '\*' << b << " fa " << prodotto(a, b) << '\n'  
<< a << '/' << b << " fa " << quoziente(a, b) << '\n';}*Da questo breve programma si imparano molte cose, ossia:

* I segni grafici (operatori) da usare per le quattro operazioni aritmetiche.
* La funzione *main* può avere degli argomenti, anzi, così com'è stata scritta, le vengono trasferiti tutti i parametri che può legittimamente ricevere come propri argomenti, nel numero e nel tipo dovuti. Senza addentrarci qui sul significato dei tipi, basterà notare che il secondo argomento (*args*) è qui usato per comunicare al programma i valori dei due operandi di cui si vogliono calcolare somma, differenza, prodotto e quoziente. Digitando, all'atto dell'esecuzione  
    
           ./a.out 12.75 48.44  
    
  i due valori indicati saranno ricevuti, come stringhe di caratteri, rispettivamente in *args[1]* e *args[2]* e trasformati nei rispettivi valori numerici dalla funzione *atof*, dichiarata e descritta al compilatore in *cstdlib*. Per il significato dei numeri racchiusi tra quadre attendete di crescere.
* È sempre opportuno che la funzione *main*, quantunque sia sempre la prima a essere eseguita, sia l'ultima a essere scritta nel documento contenente il programma: in questo modo è quasi sicuro che tutte le eventuali altre funzioni siano già note al compilatore, o perché già scritte nel documento stesso, come le nostre quattro, o perchè presenti in qualche documento incluso, come *atof*.
* Ogni altra funzione diversa da *main* le è sintatticamente equivalente: deve sempre avere, nell'ordine, un tipo, un nome, una lista di argomenti tipizzati e un ambito: così per l'appunto sono scritte *somma*, *differenza*, *prodotto* e *quoziente*.
* Quando, durante l'esecuzione del programma, in *main* o in qualsiasi altra parte si incontra il nome di una funzione nota al compilatore, ovviamente seguito da appropriati parametri racchiusi in parentesi tonde, questa viene eseguita a sua volta: ciò significa semplicemente che, al momento in cui si incontra il nome della funzione, si passa all'interno del suo ambito e si eseguono ordinatamente le istruzioni ivi contenute finché
  + o si arriva alla fine dell'ambito della funzione
  + o vi si incontra l'istruzione *return*

In entrambi i casi il programma ritorna a eseguirsi ordinatamente dal punto immediatamente successivo a quello in cui era stato incontrato il nome della funzione.

* Affinché quanto detto al punto precedente accada effettivamente occorre che vi sia corrispondenza/compatibilità di tipo tra i parametri inseriti nella lista racchiusa tra parentesi tonde allorché si vuol trasferire l'esecuzione alla funzione e gli argomenti che la funzione stessa attende; in difetto di ciò il compilatore segnala errore e NON produce il codice binario eseguibile. Non ha invece alcuna importanza il nome attribuito ai parametri rispetto al nome con cui la funzione battezza internamente i propri argomenti. Infatti in *main* sono usati i nomi *a* e *b* che non compaiono in nessun'altra parte del programma.
* Quando una funzione esegue l'istruzione *return*, a tutti gli effetti è come se in *main*, o comunque nella parte del codice da cui la funzione era stata eseguita, la richiesta della sua esecuzione venisse rimpiazzata con il valore dell'espressione che si trova a destra dell'istruzione *return* medesima. Tale espressione deve avere corrispondenza/compatibilità di tipo con il tipo cui è stata attribuita la funzione, ossia quello che ne precede il nome: questo dovrebbe bastare a spiegare come fa a funzionare il programma sopra codificato.  
    
  **POSTILLA**:   
    
  Le funzioni, in C++, sono di solito usate per scopi molto più intelligenti.

**Canto primo: tutto deve essere dichiarato**

Quando si comincia a scrivere un programma in C++ ci si deve mettere in testa che il compilatore conosce SOLO le parole del proprio vocabolario e, peraltro, PRETENDE che quelle parole siano usate in modo rigorosamente appropriato.  
Ne segue, come logica conseguenza, che qualsiasi parola, estranea al vocabolario del linguaggio, che sia introdotta nel codice, PROVOCA UN ERRORE in compilazione, non potendo essere compresa in alcun modo, A MENO CHE COMPAIA, PER LA PRIMA VOLTA, IN UNA DICHIARAZIONE.  
In altre parole le dichiarazioni sono lo strumento INDISPENSABILE per poter introdurre nel codice dei nomi, altrimenti sconosciuti, che siano atti a denominare delle entità di varia natura che il programmatore decida di voler utilizzare per perseguire i propri scopi.  
Il seguente codice è errato e NON PUÒ QUINDI ESSERE COMPILATO (provateci!)

#include <iostream>  
using namespace std;  
  
int main(  ) {  
cout << b << '\n';}

perché il compilatore non sa dare un significato all'*entità* chiamata *b*.  
  
Effettuare una dichiarazione significa attribuire un tipo, e **UNO SOLO**, all'*entità* dichiaranda, come si è già visto all'esordio della commedia, pur senza che vi si ponesse l'enfasi che sarebbe stata plausibile. Riferendosi al segmento di codice precedente, prima anche solo di citare l'*entità* ***b***, sarebbe stato necessario dichiararla, ossia attribuirle un tipo, ossia, ad esempio, scrivere una linea di codice che recitasse

int b;

che, appunto, attribuisce il tipo *int* a *b*, rendendola, da questo punto in poi, citabile e utilizzabile in ogni contesto in cui possa legittimamente apparire una quantità rappresentabile come numero intero. Ecco una versione COMPILABILE, tuttora, comunque, male scritta, del codice precedente; provate a compilarla, eseguirla e a capire perché è male scritta:

#include <iostream>  
using namespace std;  
  
int main(  ) {  
int b;  
cout << b << '\n';}

Ogni dichiarazione, una volta effettuata, VALE SOLO NELL'AMBITO IN CUI È STATA FATTA, vale a dire che NON SI ESTENDE ALL'ESTERNO DELLA PIÙ VICINA COPPIA DI PARENTESI GRAFFE che la contiene. Paradossalmente la seguente variante del precedente codice, ovviamente del tutto AUTOLESIONISTA, cagiona novamente errore (provate!):

#include <iostream>  
using namespace std;  
  
int main(  ) {  
{int b;}  
cout << b << '\n';}

Un ambito, come appare chiaro nell'ultimo codice, ne può contenere altri; se, come detto, una dichiarazione NON ESCE dall'ambito in cui si trova, viceversa ENTRA IN TUTTI GLI AMBITI AD ESSO INTERNI; il che significa che la seguente ulteriore variante TORNA A POTERSI COMPILARE:

#include <iostream>  
using namespace std;  
  
int main(  ) {  
int b;  
{cout << b << '\n';}  
}

Le conseguenze logiche di quanto scritto fin qui si riassumono nei punti seguenti:

* Ogni funzione (*main* così come ogni altra) ha, per definizione, un proprio ambito entro il quale sono confinate delle dichiarazioni che VALGONO SOLO in quella funzione; ciò significa che, entro funzioni diverse, possono trovarsi dichiarazioni diverse dello stesso nome senza che vi sia ALCUNA RELAZIONE tra queste entità omonime.
* Quando un ambito ne contiene un altro, ogni dichiarazione effettuata nell'ambito interno NON È valida in quello esterno, mentre, al contrario, ogni dichiarazione effettuata in un ambito esterno VIGE ANCHE in quelli interni, fino a qualsiasi livello di nidificazione.  
  Tuttavia è lecito ridichiarare diversamente, in un ambito interno, lo stesso nome dichiarato in uno esterno: in questo caso, nell'ambito interno, VIGE COERENTEMENTE la dichiarazione locale e quella esterna diventa non più fruibile nell'ambito interno.
* Non si può dichiarare due volte lo stesso nome nello stesso ambito, neppure se si tratta di una mera RIPETIZIONE IDENTICA, a meno che si stia ripetendo la dichiarazione del nome di una funzione.
* E a proposito della dichiarazione del nome di funzioni, l'introduzione della funzione *main* nel programma, così come quella di ogni altra funzione, ex canto precedente, È ESSA STESSA UNA DICHIARAZIONE poiché attribuisce a *main* il tipo *int* e non è confinata da alcuna coppia di parentesi graffe: quelle che seguono delimitano, come detto, l'ambito di *main*, ma la frase *int main(  )* si trova AL LORO ESTERNO; si parla, a tal proposito di AMBITO GLOBALE, esteso per tutto il documento che contiene l'intero codice.  
  In esso si trovano dichiarate appunto le funzioni che concorrono al programma e che quindi risultano COERENTEMENTE già dichiarate quando le si citi dall'INTERNO dell'ambito di una funzione scritta SUCCESSIVAMENTE nel documento.   
  Peraltro, nell'ambito globale si possono dichiarare anche entità diverse dalle funzioni e delle quali si parlerà ampiamente nei diversi canti di questo percorso.
* Tra le più interessanti di tali entità, e di cui si sta facendo uso fin dall'inizio, sono i cosiddetti *namespace* (*std* è uno di questi, evidentemente), ossia interi ambiti creati volontariamente dal programmatore e opzionalmente nidificati uno nell'altro così che in simili *matrioske* valgano le regole che si sono enunciate in questo stesso elenco.  
  Tali *namespace* possono avere un nome, come appunto *std*, o anche essere anonimi; possono essere o no qualificati con la parola di vocabolario *inline* e possono essere resi bersaglio della parola di vocabolario *using*, come si è già visto con la linea di codice *using namespace std;*. Non occorre che andiate ORA a cercare questi dettagli nella pagina dedicata.  
  Ogni *namespace* contiene dichiarazioni e SOLO dichiarazioni; null'altro, salvo il fatto che la dichiarazione di una funzione in un *namespace* può anche essere accompagnata, opzionalmente, dalla sua definizione, ossia dal suo proprio ambito.  
  Per accedere a un'entità dichiarata in un *namespace* che non sia stato sottoposto all'azione di *using* occorre prefissarla con l'intero percorso dei *namespace* che conducono alla dichiarazione, seguiti ciascuno da un'occorrenza dell'operatore *::*, denominato operatore di risoluzione di ambito; ad esempio, per accedere all'intero *x* dichiarato come qui appresso a partire dall'ambito globale:   
    
  *namespace Esterno  
  {  
    
  namespace Intermedio  
  {  
    
  namespace Interno  
  {  
    
  int x;  
  }  
    
  }  
    
  }*   
    
  occorre scrivere, ovunque ricorra la necessità,  
    
  *Esterno  :: Intermedio  :: Interno  :: x*   
    
  Va detto che una nidificazione di *namespace* deve necessariamente essere ristretta all'ambito globale, e non può essere inserita entro l'ambito di una funzione come, ad esempio, *main*.

Per concludere questo canto si sottolinea che il compilatore tratta le lettere minuscole e quelle maiuscole come LETTERE DISTINTE, il che significa che le due seguenti dichiarazioni,

int a;  
int A;

anche se effettuate nello stesso ambito, sono LEGITTIME perché, dichiarando due nomi DIVERSI, non contraddicono la terza voce del precedente elenco.  
  
Inoltre qui si anticipa, e vi si tornerà dettagliatamente in futuro, che i tipi attribuibili a un'entità all'atto della sua dichiarazione, e di cui qui il tipo *int* è stato preso come campione, sono NON SOLO tutti quelli che il compilatore conosce per propria natura, MA ANCHE tutti quelli che il programmatore può CREARE (GRAN LINGUAGGIO) di propria iniziativa.

**Canto secondo: tutto deve essere inizializzato**

Finché un'entità correttamente dichiarata resta SOLO dichiarata, si producono programmi che sono sì compilabili ed eseguibili, ma che tuttavia restano del tutto INUTILI, come nel canto precedente, allorché si sottolineava che erano comunque male scritti.  
Occorre che OGNI ENTITÀ DICHIARATA assuma un valore coerente col tipo che le è stato attribuito nella dichiarazione: ciò è quanto significa il termine tecnicistico *inizializzazione*.   
  
Si potrebbe riassumere il tutto con l'affermazione: **una dichiarazione è l'attribuzione di un tipo; un'inizializzazione è l'attribuzione di un valore (coerente)**.  
  
Va tuttavia sottolineato che mentre, per una data entità, la dichiarazione avviene UNA TANTUM in un certo ambito, di inizializzazioni nello STESSO ambito ne possono sussistere AD LIBITUM, salvo il caso in cui l'entità in questione sia qualificata come una costante.  
  
Esistono numerose maniere per effettuare un'inizializzazione; allo stato attuale della vostra conoscenza vi bastino le due seguenti che, per l'appunto, possono essere arbitrariamente ripetute; quando la vostra sapienza aumenterà vi potranno essere dette anche le altre, che però resteranno legate alla dichiarazione e saranno pertanto irripetibili:

* per assegnamento, utilizzando l'omonimo operatore, vale a dire attribuendo direttamente all'entità inizializzanda il risultato di un calcolo o dell'esecuzione di una funzione situati nel programma stesso;
* per lettura del valore da attribuire all'entità inizializzanda, estraendolo da una *sorgente di dati* esterna al programma detta, in termini tecnici e anglofoni, *input stream*.

Entrambi questi modi di inizializzazione sono già stati incontrati nel PRIMO programma proposto in questa guida e si continueranno a incontrare a ogni pie' sospinto.

**Canto terzo: l'operatore di assegnamento**

Si indica col carattere  
  
                 **=**   
  
quello che si è abituati a chiamare uguale e che, di fatto, si continua a leggere così quando lo si incontra in un programma C++, ma OCCORRE ASSOLUTAMENTE METTERSI IN TESTA che NON HA NULLA DA SPARTIRE col concetto di uguaglianza dell'analisi matematica, se non per il fatto che, dopo l'operazione da esso compiuta, i suoi due operandi hanno lo STESSO valore.   
Serve, come detto nel canto scorso, a effettuare una delle due forme di inizializzazione ed è bene cominciare ad abituarsi all'idea che si tratta di un OPERATORE che, in quanto tale, compie un'OPERAZIONE sui suoi DUE OPERANDI che gli stanno OBBLIGATORIAMENTE uno a sinistra e uno a destra. La sintassi da adottare è pertanto, rigorosamente e senza eccezioni,

*operando\_sinistro*    =    *operando\_destro*

Che cosa siano i due operandi è appunto tema del presente canto; nella riga precedente la notazione corsiva serve a evidenziare che nessuno dei due nomi utilizzati corrisponde a un'entità dichiarata, e del resto la riga stessa non è, ancora, corretto codice C++.  
  
Dato che l'operazione compiuta, come detto, è un'inizializzazione, risulta subito evidente la differenza sostanziale rispetto all'uguaglianza matematica: questa è simmetrica e quindi invariante rispetto all'ordine dei due *membri* dell'uguaglianza stessa. Fin dall'asilo sappiamo che, se *a=b*, allora è anche *b=a*.  
  
Un'inizializzazione, al contrario, NON È AFFATTO SIMMETRICA perché UNO SOLO dei due operandi, ossia *operando\_sinistro*, è quello che deve essere inizializzato; l'altro serve solo a ottenere il valore da assegnare al primo. Ne segue LOGICAMENTE che scrivere una cosa come  
*operando\_destro* = *operando\_sinistro*  
NON SOLO STRAVOLGE LA TOPOLOGIA di destra e sinistra, ma è ERRATO sul piano concettuale e addirittura può essere ERRATO sul piano sintattico, come si vedrà appresso.  
  
Dall'ultimo capoverso si evince anche, per pura DEDUZIONE LOGICA, la diversa natura dei due operandi; abituatevi a usare la LOGICA.

* Dato che l'*operando\_destro* deve fornire il valore con cui inizializzare l'*operando\_sinistro*, DEVE PER FORZA essere un'espressione calcolabile **O** la richiesta di esecuzione di una funzione che restituisca un valore appropriato; e se un'espressione deve essere calcolabile DEVE PER FORZA coinvolgere SOLO entità che siano già state TUTTE inizializzate a loro volta.
* Dato che l'*operando\_sinistro* è quello che deve essere inizializzato DEVE PER FORZA essere un'entità CAPACE DI TRATTENERE il valore fornito dall'altro operando, il che significa che DEVE corrispondere a una LOCAZIONE FISICA PERMANENTE della memoria del calcolatore in cui il valore inizializzante proveniente da destra possa essere INDELEBILMENTE SCRITTO. Allo stato attuale delle vostre conoscenze DEVE ESSERE, per farla breve, un'entità dichiarata.

Esempi di corrette inizializzazioni compiute con l'operatore =

1. Inizializzazione contestuale con la dichiarazione:

int k = 20;

a *k* si attribuisce il tipo *int* (dichiarazione) e immediatamente lo si inizializza col valore coerente 20. Non ci si dimentichi il punto e virgola finale.

1. Inizializzazione successiva alla dichiarazione:

int k = 20, c;  
// molte altre linee di programma  
c = 17 - 3 \* 5 + k;

*c* viene inizializzato col risultato (22) di un'espressione calcolabile; si noti che in questa espressione *k* è già inizializzato.

1. Inizializzazione tramite esecuzione di una funzione, magari più complicata...:

int funz(  ) {return 12;}  
int main(  ){  
int k;  
k = funz(  );  
}

*k* viene inizializzato col valore (12) restituito dall'esecuzione della funzione *funz*.

In TUTTI gli esempi sopra riportati l'inversione dei due operandi comporta evidenti errori sintattici dato che né il numero *20*, né l'espressione *17 - 3 \* 5 + k* né l'invocazione della funzione *funz(  )* potrebbero essere VALIDI OPERANDI SINISTRI non avendo NESSUNO la rappresentatività di una locazione permanente della memoria del calcolatore.   
Esistono però anche casi in cui il compilatore non può segnalare errori sintattici, e che tuttavia sono ugualmente sbagliati sul piano concettuale, come nel seguente esempio:

int k = 20, c;  
k = c;

qui l'errore non è sintattico, ma concettuale poiché *c* non è inizializzato e quindi non è un buon operando destro. Il compilatore non segnala errore, ma l'effetto ottenuto è quello di aver PERSO la corretta inizializzazione di *k* che, dopo l'operazione *k=c*, si trova bensì inizializzato, ma con un valore del tutto aleatorio che non è di certo quello che il programmatore voleva. Probabilmente intendeva piuttosto *c=k;*...

**Canto quarto: inizializzazione tramite lettura da sorgente esterna**

L'argomento riguardante la lettura di dati da parte di un programma è molto ampio e vi si tornerà sopra a tempo debito.  
Dato che in questo canto ci interessa solo l'aspetto inizializzante delle operazioni di lettura, ci limiteremo a considerare la situazione più semplice che, del resto, è già stata introdotta nel PRIMO programma di questa guida, rimandando le argomentazioni sottili come indicato poche righe fa; si era visto che la sorgente di dati esterna al programma era colà la tastiera del calcolatore.  
  
Anche in tale caso semplice, come nel canto precedente, l'inizializzazione è da considerare un'operazione, per la quale occorrono un operatore e due operandi, ma stavolta l'entità inizializzanda è l'operando di destra, ferma restando la sua natura, già descritta nel precedente canto, mentre l'operando di sinistra è l'OGGETTO *cin*, dichiarato, come è necessario che sia, nel documento *iostream* che si include all'inizio del codice, e l'operatore da usare, denominato tecnicamente *estrattore*, ha il segno grafico >>, ossia due segni consecutivi di maggiore; per inciso si cominci a prendere familiarità con la parola oggetto.  
  
Pertanto la sintassi per la corretta inizializzazione compiuta tramite lettura di un valore dalla tastiera ha il seguente aspetto:

# include <iostream>  
using namespace std;  
int main() {  
int k; // dichiarazione  
cin >> k; // inizializzazione  
// il resto del programma  
}

Si ricorda che l'inclusione del documento *iostream* è irrinunciabile, pena il fatto che *cin* risulti un'entità NON DICHIARATA, mentre la linea *using namespace std;* è facoltativa, ma, se la si omette, l'oggetto *cin* andrebbe scritto con la sua DENOMINAZIONE COMPLETA ossia come

        std::cin

Si ricorda anche che tutto quanto si scrive dopo due segni consecutivi di barra di divisione, compresi i segni stessi, viene ignorato dal compilatore fino alla fine della riga, pertanto il codice precedente è perfettamente compilabile così come si trova. Naturalmente, affinché l'inizializzazione di *k* vada a buon fine, CI VUOLE QUALCUNO VIVO E INTELLIGENTE davanti alla tastiera, e dicendo intelligente si vuol significare che si intende prescindere da eventuali balordaggini che un *minus habens* potrebbe digitare.  
  
Per quanto è stato ripetutamente detto sulla natura dell'*inizializzando* nel canto precedente, dovrebbe essere palese che le seguenti linee generano ciascuna un errore sintattico:

cin >> 20;  
cin >> 3 \* 4 - k;

**Canto quinto: inizializzazioni incoerenti**

Si è detto che, in condizioni ideali, l'inizializzazione va fatta attribuendo valori coerenti col tipo indicato nella dichiarazione e nei precedenti canti si è sempre stati ligi a questa prassi; ed è molto meglio continuare a esserlo.  
Tuttavia, talvolta, il compilatore accetta anche inizializzazioni incoerenti, segnalandole o no, secondo il livello di incoerenza e secondo il livello di pignoleria richiesto al compilatore, con diagnostici che vanno da un semplice monito, e in tal caso il codice eseguibile viene comunque prodotto, a un vero e proprio errore che impedisce la generazione dell'eseguibile. Ad esempio, delle tre seguenti inizializzazioni incoerenti:

int k = 3.14;   
double s = k;   
char c = "c";

solo la terza è considerata un errore, e come tale vieta che sia prodotto l'eseguibile; capiremo perché quando si parlerà più in dettaglio di stringhe di caratteri. La seconda, invece, è accettata tranquillamente e la prima è segnalata tutt'al più come avvertimento della perdita della parte decimale del numero 3.14.  
Quello che dunque accadrà, una volta rimossa la terza linea, sarà che tanto *k* quanto *s* saranno inizializzati con il valore 3.  
  
Anche nell'inizializzazione per lettura può accadere qualcosa di analogo, anche se, in questo caso, l'incoerenza è di gran lunga più pericolosa; se, ad esempio, di fronte alle due linee seguenti, che, va detto, il compilatore non ha motivo di disapprovare:

int k;  
cin >> k;

un eventuale orango seduto al calcolatore digitasse sulla tastiera *33.88888888888*, terminando con la pressione del tasto Invio o Enter che dir si voglia, *k* sarebbe sì inizializzato col valore 33, ma nella sorgente di dati esterna resterebbero tutti i successivi tasti digitati, e NON utilizzati, ossia .88888888888, che, con probabilità prossima a 1, finiscono per inquinare gravemente la sorgente stessa, rendendola quasi certamente VELENOSA per eventuali ulteriori letture che dovessero essere necessarie proseguendo l'esecuzione.

**Prima pausa di riflessione**

Dopo aver riletto con attenzione tutti i canti fin qui proposti (dal canto 0 al canto 5), e dopo aver esaminato, compreso e inserito nel codice le seguenti dichiarazioni con annessa inizializzazione coerente, che introducono ed esauriscono TUTTI I TIPI INIZIALIZZABILI che il compilatore conosce per suo conto, ossia senza dover attendere che il programmatore li CREI

bool b = true;   
char c = 'c';  
signed char sc = 'c';  
unsigned char uc = 'c';  
double d = 1.4142;  
long double ld = 1.4142L;  
float f = 1.4142F;  
int i = 1;  
signed int si = 1;  
unsigned int ui = 1;  
long int li = 1L;  
signed long int sli = 1L;  
unsigned long int uli = 1UL;  
long long int lli = 1LL;  
signed long long int slli = 1LL;  
unsigned long long int ulli = 1ULL;  
short int shi = 1;  
signed short int sshi = 1;  
unsigned short int ushi = 1;  
void v(); // NON INIZIALIZZABILE!

provate a scrivere da voi e a eseguire un programma (aiutandosi anche con quello proposto nel canto zero) che faccia le seguenti cose, NEL CITATO ORDINE:

* scriva sul terminale il quadrato di *f* (usate l'oggetto *cout*, come nel canto zero)
* richieda da tastiera un nuovo valore per *i*, che lo reinizializzi in base a quello che digiterete;
* usando questo nuovo valore di *i*, calcoli i risultati delle somme di *i* stesso con *slli*, *d* e *ld* e li scriva sul terminale;
* esegua, in base a quanto già conoscete, ogni altra operazione che la vostra fantasia vi suggerisca.

Si tengano presenti le seguenti annotazioni:

* in tutte le numerose dichiarazioni che terminano col tipo *int* il qualificatore *signed* è implicato dalla sua assenza, il che rende funzionalmente IDENTICHE, ad esempio, le dichiarazioni di *i* e *si*;
* la stessa cosa NON accade per le dichiarazioni che terminano col tipo *char*, per cui, passando da un compilatore a un altro, non si può sapere se la dichiarazione di *c* sia funzionalmente identica a quella di *sc* o a quella di *uc*;
* la parola *int* stessa può essere omessa quando la dichiarazione è accompagnata da altre specifiche (*signed*, *short*, *long*...)
* i suffissi alfabetici usati negli operandi destri di certe immediate inizializzazioni (e che potrebbero essere indifferentemente scritti anche in minuscole) sono quelli che servono affinché il compilatore riconosca la PERFETTA COERENZA dell'inizializzazione stessa; in pratica, in base a quanto scritto nel precedente canto, quasi mai si mettono e, in particolare, NON SI DEVONO METTERE se l'inizializzazione avviene tramite lettura;
* nelle dichiarazioni in cui il tipo consta di più di una sola parola, ognuna commuta con tutte le altre;
* al tipo *void* non può essere attribuito alcun valore; per questo è stato usato solo per dichiarare una funzione che non restituisca nulla.

**Canto sesto: c'è chi è costante e chi no**

Fin qui si è sempre parlato di entità a proposito dei contenuti di un programma; un termine talmente generico da non aver bisogno di ulteriori caratterizzazioni.  
Da questo canto in poi si comincerà a parlare di variabili e di costanti, in attesa di poter parlare di oggetti, e si abbandonerà definitivamente il troppo indefinito termine *entità*.  
  
Cominciamo col dire che una variabile è ciò che NON PUÒ ESSERE DEFINITO UNA COSTANTE e che, tuttavia, è in grado di detenere un valore per una certa durata temporale, cosa che le costanti fanno perennemente.  
  
Le costanti, a loro volta, possono essere raggruppate in tre grandi categorie:

* costanti esplicite
* costanti definite
* costanti dichiarate

Le costanti esplicite entrano nel programma DIRETTAMENTE COL LORO VALORE: il numero intero ***20***, il numero decimale ***33.78***, la stringa, delimitata da virgolette, ***"tanto va la gatta al lardo"***, il carattere, delimitato da apostrofi, ***'A'***, sono tutti esempi di costanti esplicite che sono state già incontrate numerose volte fino a qui.  
Per loro stessa natura non hanno bisogno di essere dichiarate: il tipo cui appartengono è evidente di per sé e del resto chi vedrebbe l'utilità di una dichiarazione come *int 20;* ? Nè, tanto meno, è consentito inizializzarle: una linea che recitasse *20 = 10 + 10;* sarebbe sbagliata nonostante dica il vero, e questo a conferma del fatto che l'operatore di assegnamento NON È l'uguale della matematica.  
A dispetto della loro apparente comodità e della loro immediatezza, si può affermare con giusta ragione che un programma è tanto meglio scritto quanto minore è il numero di costanti esplicite che contiene. I diversi modi di introdurre nel codice una costante esplicita, secondo i diversi tipi cui può appartenere, sono dettagliatamente descritti in una pagina apposita del sito; per adesso vi può bastare quel che è stato mostrato poche righe fa.  
  
Le costanti definite sono percepite dal compilatore come costanti esplicite e dal programmatore come variabili; la ragione di questa apparente contraddizione risiede nel fatto che a definirle è il preprocessore, e quindi, come per le costanti esplicite, non occorre che le si dichiari né si può inizializzarle. Quel che succede è che una linea di codice che appare scritta *x = venti;*, ove *x* è dichiarata dal programmatore, mentre *venti* è definita dal preprocessore, in realtà viene letta dal compilatore come *x = 20;* dando per scontato che la definizione operata risenta in modo coerente del nome usato per portarla a compimento, ossia che al preprocessore sia stato fatto eseguire un *#define venti 20*  
Di questa categoria di costanti, allo stato attuale della vostra conoscenza, potete tranquillamente fare a meno: giusto sappiate, per ora, che esistono...  
  
Le costanti dichiarate, come dice la parola stessa, devono essere dichiarate e, PER FORZA immediatamente inizializzate perché la loro inizializzazione avviene UNA TANTUM esattamente come accade per la dichiarazione, e quindi si verifica nello stesso momento, altrimenti che costanti sarebbero? La sintassi da usare è, utilizzando sempre il tipo *int* come campione:

const int voglio\_dichiarare\_una\_costante = 27;

si notino il qualificatore *const* e l'immediata inizializzazione. Da tale riga, e per tutto l'ambito in cui è posta, non solo *voglio\_dichiarare\_una\_costante* non potrà mai più essere inizializzata, ma non potrà neppure apparire in alcuna espressione che abbia la potenzialità di modificarla: in ognuno di questi casi il compilatore segnalerebbe errore. Potrà invece apparire in qualsiasi operando destro che non abbia effetti collaterali su di essa, apportando il proprio contributo costante alla determinazione del valore finale dell'operando stesso.   
Si noti che la stessa omissione dell'immediata inizializzazione sarebbe segnalata come errore, dato che, in tal caso, per quello che si è appena detto, la sola dichiarazione sarebbe ASSOLUTAMENTE INUTILE. L'interscambio delle due parole *const* e *int* è accettato ed equivalente.   
  
Quantunque non sia del tutto pertinente col titolo del canto presente, lo si concluderà, prendendo spunto dalla doverosa immediata inizializzazione della costante dichiarata *voglio\_dichiarare\_una\_costante*, introducendo un'altra entità (e questa è DAVVERO l'ultima volta che si userà questa parola) che abbisogna di immediata inizializzazione all'atto della dichiarazione, pur non essendo necessariamente una costante: il cosiddetto riferimento sinistro in memoria (in angloamericano: *lvalue reference*). Si consideri il codice seguente:

# include <iostream>  
using namespace std;  
  
int main( )  
{  
int k = 1, l = 2;  
int &j = k;  
cout << j << '\n';  
j = l;  
cout << j << '\n';  
}

e si presti attenzione alla dichiarazione di *j*, il cui nome è prefissato col segno grafico &: *j*, così dichiarato, NON È una semplice variabile intera, come lo sono *l* e *k*, ma appunto un riferimento sinistro a una cella di memoria contenente una variabile di tale tipo, preso sempre come campione, e OBBLIGATORIAMENTE E PERFETTAMENTE COERENTE con l'inizializzazione. Il codice proposto non consente di apprezzare pienamente la differenza, ma nel seguito si vedrà che si tratta di una diversità sostanziale. Per adesso l'unica differenza che si osserva chiaramente consiste nel fatto che, se *j* fosse stata una variabile qualsiasi, la sua inizializzazione si sarebbe potuta rinviare rispetto alla dichiarazione, piuttosto che essere obbligatoria: provate a toglierla, oppure a renderla APPENA un pochino incoerente, e vedrete che il compilatore IMMEDIATAMENTE se ne dorrà molto.  
  
Accanto ai riferimenti sinistri, ci sono anche (potevano mancare?... in effetti sono stati introdotti solo con la versione 2011 del linguaggio) i riferimenti destri (*rvalue reference*), anch'essi da inizializzare immediatamente all'atto della dichiarazione e per la cui comprensione i tempi non sono ancora maturi; qui si darà solo l'esempio sintattico della dichiarazione di uno di loro, sempre utilizzando il tipo *int* SOLO come campione, giusto affinché sappiate FIN DA ADESSO che esistono e osserviate COME sono inizializzati:

# include <iostream>  
using namespace std;  
  
int main( )  
{  
int k = 1, l = 2;  
int &&j = std :: move(k);  
cout << j << '\n';  
j = move(l);  
cout << j << '\n';  
}

Si osservi il raddoppiamento del segno grafico & SENZA SPAZI frammezzo e la necessità di invocare, per la corretta inizializzazione, la funzione *move*, definita nel *namespace std*, come sottolineano volutamente i due modi diversi usati per richiederne l'esecuzione.  
Se eseguirete questo codice (**FATELO!**) vedrete che produce un risultato IDENTICO a quello precedente, ma quel che è avvenuto nella memoria del calcolatore è COMPLETAMENTE DIVERSO, ed è ciò che non siete ancora pronti a comprendere.  
  
Si possono dichiarare riferimenti, sia sinistri sia destri, ovviamente con le dovute e corrette inizializzazioni, come indicato negli esempi, praticamente a OGNI COSA, anche a quelle che ancora non conoscete a fondo o non conoscete affatto, come *array*, puntatori, funzioni e anche altri riferimenti: tenetelo presente quando incontrerete ciascuna di queste componenti del linguaggio.  
In ognuno di tali contesti la dichiarazione di un riferimento appropriato dovrà obbedire alle necessarie regole sintattiche, tenendo presenti i seguenti fatti, peraltro ovvii nella loro stessa semantica:

* **un riferimento a funzione NON è la stessa cosa di una funzione che restituisce un riferimento. Entrambi questi enti esistono indipendentemente e possono essere dichiarati entrambi.**
* **un riferimento ad *array*, che esiste, come si è detto, NON è la stessa cosa di un *array* di riferimenti che è un ente INESISTENTE, così come INESISTENTE è, per inciso, un *array* di funzioni.**
* **un riferimento a puntatore NON è la stessa cosa dell'indirizzo di un puntatore: un ente che ESISTE ma certo NON in una dichiarazione; o, peggio, di un puntatore a riferimento, che NON ESISTE PROPRIO.**

Queste cose sono state già scritte, e pure in grassetto, perché crediate nel C++, anche prima di conoscerlo, e perché, credendo, abbiate la facoltà di diventare **buoni programmatori**.  
In sostanza, e per farla breve, **ficcatevele in testa da subito**.

**Canto settimo: le espressioni**

Col termine *espressione* s'intende qualsiasi segmento di programma C++ che NON SIA NÉ UNA DICHIARAZIONE NÉ UNA PAROLA DEL VOCABOLARIO DEL LINGUAGGIO; va da sé, quindi, che la parte larghissimamente preponderante di un programma C++ è costituita di espressioni.  
Fino a questo punto ne sono già state incontrate numerose, e quasi ognuna di esse può essere decomposta in *sottoespressioni*, vale a dire in null'altro che espressioni più semplici.  
Se tornate a leggere il primo programma proposto nel canto introduttivo, troverete, ad esempio, delle righe di codice che qui si riportano per comodità di discorso:

cout   
<< a << '+' << b << " fa " << somma(a, b) << '\n'  
<< a << '-' << b << " fa " << differenza(a, b) << '\n'  
<< a << '\*' << b << " fa " << prodotto(a, b) << '\n'  
<< a << '/' << b << " fa " << quoziente(a, b) << '\n';

Queste righe sono UNA SOLA espressione, costituita di numerose sottoespressioni: la utilizzeremo per analizzarne la forma e la sostanza.  
Sintatticamente un'espressione è obbligatoriamente chiusa dal segno di punto e virgola, tranne pochissime motivate eccezioni; inizia: **O** subito dopo il punto e virgola che chiude un'altra espressione che la precede; **O** subito dopo il punto e virgola che chiude una dichiarazione, la quale può, a sua volta, contenere anche espressioni in caso di immediata inizializzazione; **O** subito dopo una parola di vocabolario, e in tal caso occorre rispettare la sintassi propria di ciascuna parola, ed è qui che si incontreranno le poche eccezioni alla norma della fine-espressione decretata dal punto e virgola; **O** immediatamente dopo una parentesi graffa aperta, che inizia un ambito; **O** immediatamente dopo una parentesi graffa chiusa, che termina un ambito.  
  
Per quanto riguarda la nostra espressione-campione, il suo inizio, se tornate a esaminare il programma che la conteneva, ha luogo subito dopo una dichiarazione inizializzata, ossia nella seconda delle alternative elencate.  
  
Qui appresso sono introdotte le caratteristiche essenziali di un'espressione:

* Ogni espressione è formata da operandi e operatori, in numero qualsiasi;
* Ogni espressione ha UN SOLO VALORE corrispondente a un certo tipo, o NESSUN VALORE;
* Ogni espressione, se ha senso, produce effetti collaterali, utili, neutri o dannosi per il programma; quest'ultima eventualità va ovviamente EVITATA da ogni buon programmatore.

Nella nostra espressione-campione figura UN SOLO operatore, l'inseritore su *output* <<, che vi compare ripetuto ben 24 volte; tutto il resto dell'espressione è costituito di operandi, di tipi diversi: costanti stringa, costanti carattere, variabili dichiarate, invocazioni di funzioni e l'oggetto *cout*. Si può dire che OGNI SINGOLO OPERANDO è, a sua volta, un'espressione irriducibile, come parziale, ovvia, eccezione alla prima voce del precedente elenco.  
  
Analizzando più dettagliatamente l'espressione, procedendo da sinistra a destra, si incontra una prima *sottoespressione*, costituita da *cout << a*, in cui compare una sola occorrenza di un operatore.  
Chi siano qui l'operatore e gli operandi è già stato detto, ed è piuttosto evidente che l'unico effetto collaterale di questa sottoespressione consiste nell'invio del valore di *a* sul canale di *output* e quindi nella comparsa di tale valore sul terminale: si tratta di un effetto utile al programmatore, ma del tutto neutro per il programma.  
  
Ma QUAL È IL VALORE di questa sottoespressione? Se si trattasse di un'espressione irriducibile, costituita SOLO di un operando, il suo valore sarebbe quello dell'operando, senza ulteriori discussioni, ma qui che valore ha? E a quale tipo appartiene tale valore?  
  
La regola generale è che il valore di un'espressione elementare, definendo così una contenente il numero minimo plausibile di operatori, ossia UNO e UNA SOLA VOLTA, è determinato dal tipo di operatore coinvolto: nel nostro caso l'operatore *<<* attribuisce, come valore, all'espressione che lo contiene il suo operando sinistro stesso, ossia l'oggetto *cout* e il tipo è pertanto quello di questo stesso oggetto. Si potrebbe affermare che l'espressione *cout << a* vale *cout*...e questo basta a spiegare come funziona tutto il resto dell'espressione: se l'affermazione è vera, come è vera, allora dopo la scrittura del valore di *a* (effetto collaterale) è come se ci si trovasse di fronte, come sottoespressione seguente, a *cout <<'+'* e così via, ricorsivamente, fino alla fine dell'intera espressione, il cui valore finale, non più utilizzato, è ancora *cout* medesimo.  
Un altro operatore che attribuisce all'espressione in cui si trova il valore del proprio operando sinistro è l'operatore di assegnamento, per cui un'espressione come

x = y = 2;

ha come VALORE 2 e come EFFETTO COLLATERALE, non si sa se utile o dannoso, data la mancanza di contesto, il fatto che tale valore sia assunto anche da entrambe le variabili dichiarate *x* e *y*.  
Altri operatori hanno comportamenti diversi: ad esempio quelli aritmetici attribuiscono all'espressione in cui si trovano il valore che ognuno si aspetterebbe, come accade, ad esempio, nell'espressione *x+y;*.  
Si osservi, peraltro, che quest'ultima espressione, presa da sola, NON HA ALCUN SENSO perché NON HA EFFETTI COLLATERALI; ciò non significa che sia considerata errata. In altre parole le linee seguenti

int x = 1, y = 2; // dichiarazione inizializzata  
x+y;   
// espressione sintatticamente corretta, di valore 3,  
// ma senza alcun effetto collaterale

sono compilate correttamente, dato che non contengono alcun errore sintattico, ma sono totalmente prive di significato, come suggeriscono anche i commenti. Invece ha senso pieno la seguente variante:

int x = 1, y = 2; // dichiarazione inizializzata  
int z; // dichiarazione non inizializzata  
z = x+y;   
// espressione sintatticamente corretta, di valore 3,  
// effetto collaterale: inizializzazione di z con 3

Si è detto che un'espressione può NON AVERE ALCUN VALORE: si parla, in tal caso, di espressione NON VALUTABILE. Facendo appello alla logica si desume immediatamente che una tale espressione non potrà MAI essere una sottoespressione di un'altra espressione, cui non sarebbe in grado di apportare alcun contributo, e di fatto se ciò accadesse il compilatore lo segnalerebbe come errore. Dovrà quindi PER FORZA essere un'espressione irriducibile, ossia non contenere alcun operatore salvo, tutt'al più, l'operatore virgola (vedi appresso). Ma, in tal caso, non avendo valore, dovrà PER FORZA avere almeno effetti collaterali, altrimenti sarebbe più inutile di un *sali e tabacchi* sulla sponda del mar Morto e più cervellotica delle moviole sulla stampa sportiva italiana.   
Pertanto l'unica espressione irriducibile e SENZA VALORE che ha ancora significato è l'invocazione dell'esecuzione di una funzione dichiarata *void* come nel seguente esempio:

# include <iostream>  
using namespace std;  
  
void funz(int x)   
{cout << x <<'\n';}   
  
int main(  )   
{int z; // dichiarazione  
cout << "dimmi quanto vale z ", cin >> z;  
// espressione con inizializzazione di z  
funz(z);   
// espressione SENZA valore, CON effetti collaterali  
}

In quest'ultimo codice mette conto analizzare la linea

cout << "dimmi quanto vale z ", cin >> z;

che, per quanto detto in questo stesso canto, contiene UNA SOLA ESPRESSIONE: c'è un unico punto e virgola, alla fine della riga. Questo significa che, da sintassi, la virgola va trattata alla stregua di un operatore e, di fatto, È UN OPERATORE. La sua operazione consiste nel separare una sottoespressione da un'altra all'interno di un'espressione più complessa, facendo scartare il valore della sottoespressione alla sua sinistra nella valutazione del valore finale dell'intera espressione in cui si trova.  
Ne segue, secondo LOGICA e secondo anche quanto fin qui detto, che l'espressione in questione ha come effetti collaterali, nell'ordine:

* la comparsa sul canale di *output* del messaggio tra virgolette;
* la lettura di un dato digitato sulla tastiera;
* l'inizializzazione di *z* col valore di tale dato, purchè, come sempre, a digitare non sia stata una scimmia.

e che il suo VALORE è quello di *cin*.  
  
Per concludere questo canto verrà citato l'aureo motto noto a ogni buon programmatore C++ e che si suggerisce di scolpire nella mente in modo indelebile:  
  
**TUTTO È VERO QUEL CHE NON È ZERO**  
  
il che significa che qualsiasi espressione VALUTABILE può essere trattata come la costante booleana **true** quando il suo valore è NON NULLO e come la costante booleana **false** nel caso contrario.  
Viceversa, un'espressione NON VALUTABILE non ha neppure alcun equivalente booleano.

**Canto ottavo: i contenitori di dati - parte I: gli *array***

Tutti sanno che in Fisica esistono grandezze vettoriali (forza, velocità, posizione), tensoriali (campo elettro-magnetico, momento angolare) e via dicendo. Nei testi canonici è d'uso indicare i vettori con lettere in grassetto, o sormontate da un segno di freccia, e le singole componenti con nomi diversi; ad esempio non è infrequente trovare la seguente definizione della forza

**F** = (Fx, Fy, Fz)

La scelta di attribuire nomi diversi alle singole componenti dello STESSO vettore è **INFELICE** in C++, dato che il linguaggio prevede appunto la possibilità di dichiarare delle variabili/costanti che fungono da veri e propri *contenitori* di dati omogenei e omologhi. Di questi contenitori il più semplice è l'*array*, che è appunto un contenitore costante, nel senso che la sua capienza e la sua morfologia sono stabilite una volta per tutte all'atto della dichiarazione e non sono mai più modificabili in tutto l'ambito in cui la dichiarazione è posta.  
  
Il buon programmatore C++ che voglia introdurre nel proprio programma una forza applicata a un punto materiale non la dichiarerà dunque come

double Fx, Fy, Fz; // questa dichiarazione fa schifo

ma piuttosto come

double F[3]; // questa sì che va bene

Qui *F* è per l'appunto dichiarato essere un ***array*** del tipo *double*, ossia un contenitore di *double* atto a contenerne esattamente 3: è giusto quel che ci vuole per una forza, o per qualsiasi altro vettore della Fisica. Naturalmente, se si esula dall'interpretazione fisica, assume altrettanto valido significato una dichiarazione come questa

int i[20];

in cui si dichiara un contenitore (*array*) *i* per 20 *int* che, evidentemente, serviranno al programmatore.  
L'*array*, come si è già detto, è un contenitore costante, e questo si evince anche dal fatto che la sua capienza, ossia il numerello che la sintassi impone di inserire tra le parentesi quadre, può SOLO essere, allo stato attuale della vostra conoscenza, una **costante intera** esplicita o definita o dichiarata, NON una variabile intera; che poi ci si metta addirittura una quantità non intera è degno solo di un gorilla. Per la verità taluni compilatori, tra cui il GNU di Linux, ammettono anche l'uso di una variabile intera, ma, visto che ci sono alternative migliori, mi sento di **sconsigliarne caldamente l'uso**.  
  
Per accedere a ciascun elemento contenuto in un *array* dichiarato, per farne ciò che si desidera, si usano ancora le parentesi quadre ponendovi entro, stavolta, QUALSIASI ESPRESSIONE di tipo *int* che abbia un valore che ricada nell'intervallo rappresentativo della capienza dell'*array*: a tal proposito si sottolinea che gli elementi di un *array* si reperiscono al suo interno misurandone la distanza dall'inizio. Tanto per intendersi il PRIMO elemento dell'*array* *i*, trovandosi a distanza ZERO dall'inizio, è individuabile come *i[0]* mentre l'ULTIMO ELEMENTO è *i[19]*, NON *i[20]*, che è un ERRORE, perchè dista dall'inizio lo spazio occupato dai suoi 19 fratelli che lo precedono. Una volta abituatisi a questo, non si faticherà a comprendere che era la cosa GIUSTA da fare, e che questo sistema di reperimento è di GRAN LUNGA MIGLIORE rispetto a quello in cui si pensi di numerare da 1 a 20.  
  
Un *array* può beneficiare di immediata inizializzazione del suo contenuto, all'atto stesso della dichiarazione; naturalmente questa prerogativa è tanto più praticabile quanto più minuscola è la capienza dichiarata, altrimenti si diventa canuti precocemente; anzi, se un *array* viene immediatamente inizializzato la sua capienza può essere addirittura omessa, lasciando al compilatore l'onere di desumerla contando in proprio il numero di inizializzatori inseriti.  
Ad esempio, per inizializzare immediatamente la nostra forza *F* al valore di 2.5 *newton* lungo l'asse X e 0 lungo gli altri due assi, andrebbe bene una qualsiasi, e SOLO UNA, delle quattro seguenti linee, tra di loro del tutto equivalenti:

double F[3] = {2.5, 0.0, 0.0};  
double F[3]    {2.5, 0.0, 0.0};  
double F[  ] = {2.5, 0.0, 0.0};  
double F[  ]    {2.5, 0.0, 0.0};

Come si vede la lista degli inizializzatori va racchiusa tra graffe e ogni inizializzatore deve essere separato dal successivo tramite una virgola; gli inizializzatori devono essere ESPRESSIONI VALUTABILI di tipo compatibile con quello dichiarato per l'*array* stesso, dovendo ordinatamente inizializzare ciascun elemento contenuto. Si apprende anche che la presenza del segno = (operatore di assegnamento) è facoltativa, SOLO però a partire dallo standard 2011, e che lasciare vuota la coppia di parentesi quadre, tuttavia presenti OBBLIGATORIAMENTE, è consentito purché appunto sia data una lista inizializzante dato che, in tal caso, essendo il compilatore capace di contare, è in grado di capire da solo quanto debba essere grande l'*array* dichiarato. La capienza così autodeterminata resta poi comunque fissa e immodificabile.  
  
Un *array* può anche essere un contenitore di altri *array*, come vuol significare una dichiarazione come questa:

double a[2][3];

Qui *a* è un *array* il cui contenuto consiste nei due *array* *a[0]* e *a[1]*, come indica la prima delle due capienze dichiarate, ciascuno dei quali, a sua volta, ha capienza, la seconda, per 3 valori di tipo *double*. In sostanza *a* finisce con l'essere un contenitore di 6 *double* in tutto, il prodotto delle due capienze, organizzati in 2 gruppi di 3. Ovviamente la cosa può proseguire fino a esaurimento della memoria e/o comodità d'uso dando senso a dichiarazioni come

int b[2][4][6]; // 48 interi  
char c[4][8][2][9]; // 576 caratteri

e via dicendo.   
L'immediata inizializzazione di *array* multipli è consentita usando la medesima sintassi già introdotta; va osservato però che SOLO UNA delle capienze richieste può essere omessa e, precisamente, SOLO LA PRIMA. Usando *a* come prototipo ecco due sue possibili immediate inizializzazioni, entrambe legittime

double a[2][3] {1., 2., 3., 4., 5., 6.};

oppure

double a[  ][3]={1., 2., 3., 4., 5., 6.};

Dovrebbe essere abbastanza intuibile che i primi tre inizializzatori andranno a riempire *a[0]*, mentre i secondi tre riempiranno *a[1]*.  
Se uno ha paura di confondersi, o vuol vedere a colpo d'occhio che cosa va a finire dove, sono contemplate anche le seguenti alternative, utili,forse, al programmatore, ma del tutto trasparenti per il compilatore:

double a[  ][3]={ {1., 2., 3.}, {4., 5., 6.} };  
double a[  ][3]={ 1., 2., 3., {4., 5., 6.} };  
double a[  ][3]={ {1., 2., 3.}, 4., 5., 6. };

La prima di queste, riscritta, beneficiando del formato libero del codice, come segue:

double a[  ][3]={   
      {1., 2., 3.},   
      {4., 5., 6.}   
};

consente anche al più arretrato apprendista di capire che cosa sta succedendo.  
  
L'ultima cosa da dire a proposito della dichiarazione, con inizializzazione immediata, di un *array* è che bisogna SEMPRE privilegiare l'opzione consentita di lasciar vuota una coppia di parentesi quadre, perché la seguente linea

int x[2] = {1, 2, 3};

CAGIONA ERRORE IN COMPILAZIONE, al contrario di questa

int x[  ] = {1, 2, 3};

che viene compilata con gusto dal compilatore. La ragione è EVIDENTE nel chiaro peccato di incoerenza commesso nel primo caso, mentre nel secondo si è mostrata umiltà nel lasciar fare al compilatore. Ovviamente qualcosa come *int x[3]={1,2};* non produce errore, ma lascia NON INIZIALIZZATO l'ultimo elemento contenuto in *x*.

**Canto nono: i contenitori di dati - parte II: i puntatori**

Ogni variabile dichiarata, in qualsiasi ambito, è individuata, dal punto di vista del programmatore, dal nome che questi le ha assegnato all'atto della dichiarazione e, dal punto di vista del programma, dall'indirizzo della memoria in cui il sistema operativo del calcolatore l'ha collocata. Ogni volta che il programmatore cita il nome di una variabile dichiarata, di fatto il calcolatore accede a un preciso indirizzo di memoria per compiere operazioni che possono sfruttare il valore ivi contenuto o modificarlo, secondo la natura dell'operazione richiesta.  
DOVE sia situata la cella di memoria occupata da una certa variabile, per TUTTO il tempo della sua esistenza, ossia fino a quando l'esecuzione del programma non abbandoni l'ambito in cui è dichiarata, potrebbe essere ritenuta un'informazione superflua da parte di ogni programmatore apprendista.  
  
Almeno fino alla lettura e comprensione del presente canto.  
  
Per la verità, anche allorché si è parlato degli *array*, nel canto precedente, si è silenziosamente introdotto il concetto di conoscenza dell'indirizzo in cui certe variabili sono situate nella memoria. Infatti nella dichiarazione di un *array*, come quella che segue

int   un\_array\_che\_non\_servirebbe[1];

il nome dell'*array* (*un\_array\_che\_non\_servirebbe*), essendo quello del contenitore che ha come contenuto l'unica variabile intera *un\_array\_che\_non\_servirebbe[0]*, ne è, in pratica, l'indirizzo in memoria, e se ne potrebbe anche prendere visione eseguendo (FATELO COME ESERCIZIO!) le seguenti linee

int   un\_array\_che\_non\_servirebbe[  ] {10};   
cout   
<< un\_array\_che\_non\_servirebbe[0]   
<< " si trova in "  
<<un\_array\_che\_non\_servirebbe <<'\n';

Se siete riusciti a produrre il codice eseguibile, eseguitelo più volte: vedrete che, a ogni esecuzione, apparirà una cosa diversa a destra di *" 10 si trova in "*, a riprova del fatto che la collocazione fisica di una variabile nella memoria non influenza l'esecuzione di un programma. Provate ora la seguente variante:

int    un\_array\_utile[  ] {10, 20, 30};  
cout   
<< un\_array\_utile[0]   
<< " si trova in "   
<<un\_array\_utile << '\n'  
<< un\_array\_utile[1]   
<< " si trova in "  
<<un\_array\_utile+1 << '\n'  
<< un\_array\_utile[2]   
<< " si trova in "  
<<un\_array\_utile+2   
<<'\n';

Anche in questo caso esecuzioni successive producono risultati differenti, ma gli scostamenti relativi tra gli elementi consecutivi dell'*array* SONO FISSI e valgono 4 (quattro), vale a dire esattamente il numero di *bytes* occupati dalle variabili di tipo *int*. Tra l'altro si vede come si faccia a ottenere le posizioni in memoria degli elementi successivi al primo.  
  
Dunque l'informazione su dove siano collocate le variabili contenute in un *array* esiste, ma finché si parla di *array*, essendo tali collocazioni delle costanti di valore indipendente dalla volontà del programmatore, ne può essere fatto praticamente nulla di utile, un po' come l'energia di massa *mc^2* del nostro corpo: sappiamo che c'è, sappiamo anche quanto vale (basta pesarsi), è una costante, più o meno errori sperimentali, diete dimagranti o pranzi natalizi, ma fino a quando non ci annichiliamo con un altro *noi stesso* fatto di antimateria nessuno se ne può far niente.   
  
Le cose cambiano radicalmente, e in maniera sconvolgentemente POTENTE, quando entrano in scena i puntatori: per inciso anche gli *array* sono dei puntatori, esattamente dei puntatori costanti.  
  
Ora fate appello alla vostra LOGICA: che cosa implica l'ultimo capoverso? Se i puntatori sono uno strumento del linguaggio più POTENTE rispetto agli *array* e questi ultimi sono tuttavia puntatori costanti, significa indubitabilmente che la potenza dei puntatori risiede nella loro possibilità di non essere costanti; ma se una variabile, appunto VARIABILE, ha come valore MODIFICABILE il luogo della collocazione in memoria di un'altra, o altre, variabile/i (è QUESTO, lo si metta bene in mente, il VALORE di qualsiasi puntatore) allora significa che è in grado di indirizzare (con questo verbo si intende appunto *detenere come proprio valore l'indirizzo di...*) zone POTENZIALMENTE DIVERSE della memoria, secondo le esigenze del programmatore, e, al limite, può anche NON INDIRIZZARE UN BEL NULLA.  
  
Nelle ultime proposizioni scritte sta appunto la POTENZA dei puntatori.  
  
Come qualsiasi altra variabile, un puntatore va dichiarato e inizializzato: senza quest'ultima operazione, non solo è INUTILE come qualsiasi variabile non inizializzata, ma addirittura NOCIVO o ESIZIALE per il programma.  
  
Il compilatore capisce che si sta dichiarando un puntatore quando, in una dichiarazione, si premette il segno di asterisco, lo stesso carattere usato per indicare l'operatore aritmetico di moltiplicazione, al nome della variabile dichiarata; così, ad esempio, nella seguente dichiarazione

int  i = 7,  \*k,  r,  a[  ] {1, 2, 3};

sono dichiarate due normalissime variabili intere, *i* e *r*, la prima anche inizializzata, un *array* (*a*) di tre interi inizializzati e un puntatore a interi chiamato *k*.  
  
Per inizializzare un puntatore esistono diverse modalità che qui di seguito si elencano:

* gli si può assegnare il valore di un altro puntatore, di uguale tipo, già inizializzato, anche proveniente dal valore restituito da una funzione; rispetto alla nostra dichiarazione sarebbe lecita, come esempio di questa modalità, l'inizializzazione  
  *k = a+1;*
* gli si può assegnare il valore della posizione in memoria di una semplice variabile di tipo uguale, anche non ancora inizializzata, utilizzando, per ricavarlo, l'operatore unario *&*, DA NON CONFONDERE con lo stesso segno usato in una dichiarazione, che si è incontrato nel canto sesto: in quel contesto NON È un operatore(!); sempre avendo in mente la stessa dichiarazione, questa modalità si esprime, ad esempio, nell'inizializzazione  
  k = & r;
* gli si può assegnare il risultato dell'azione dell'operatore *new*; è questa la modalità foriera del maggior numero di esiti utili e positivi, dato che, come dice il nome stesso dell'operatore, al puntatore viene associata un'area di memoria TOTALMENTE NUOVA, vale a dire ASSENTE prima che l'operatore agisse.

Il seguente programma (COMPILATELO ED ESEGUITELO!) illustra quanto detto nel precedente elenco puntato; inoltre offre lo spunto per una discussione leggermente più approfondita sull'operatore *new*

# include <iostream>  
using namespace std;  
  
int main(  )  
{  
int i=7, \*k, r, a[  ]{10, 20, 30};   
k = &r, // 1  
r = 5005, // 2  
cout << k[0] << " si trova in " << k << '\n',  
k = &i, // 3  
cout << k[0] << " si trova in " << k << '\n',  
k = a+1, // 4  
cout << k[0] << " si trova in " << k << '\n',  
k = new int, // 5  
cout << k[0] << " si trova in " << k << '\n',  
k = new int[i/3]{99, 55}, // 6  
cout << k[0] << " si trova in " << k << '\n',  
cout << k[1] << " si trova in " << k+1 << '\n',  
// cout << k[2] << " si trova in " << k+2 << '\n',   
// ERRORE!  
k = new int(22), // 7   
cout << k[0] << " si trova in " << k << '\n',  
k = new int(r/i), // 8  
cout << k[0] << " si trova in " << k << '\n';  
}

Per prima cosa si osservi che la sintassi da usare per accedere ai valori situati nella memoria indirizzata tramite un puntatore è IDENTICA a quella già incontrata con gli *array* (repetita iuvant: sono pure puntatori...);   
altresì si osservi che il puntatore *k* viene inizializzato ben sette (7) volte, e quindi è tutt'altro che una costante;  
nella riga *//1* si inizializza *k* con la locazione in memoria di *r*, che viene inizializzato solo successivamente col valore 5005; ma questo basta a far apparire in *output* tale valore nella riga successiva, tramite *k[0]* che è il valore che si trova nella locazione indirizzata da *k*, e quindi è *r* stesso. Lo stesso vale per la successiva inizializzazione di *k* alla riga *// 3*, seguita da una sottoespressione *cout* uguale alla precedente.   
Alla riga *// 4* il puntatore viene reinizializzatto assegnandogli il valore di *a+1*: coerentemente il successivo *cout*, IDENTICO, fa apparire il valore del secondo elemento di *a*.  
  
Dalla riga *// 5* in poi entra in scena l'operatore *new* che, nella riga *// 5*, appunto, offre l'esempio del suo più semplice utilizzo; il significato dell'operazione compiuta è contenuto nella seguente frase, scritta in italiano con il soggetto che è l'operatore stesso (traduzione mia dal C++):  
  
*"cerco nella memoria del calcolatore un'area disponibile* (significa non ancora utilizzata [NdT]) *di estensione pari almeno a quello che serve a contenere un singolo* ***int****, e quando l'avrò trovata la riserverò in esclusiva a questo programma, vietando l'accesso a chiunque altro, creerò un puntatore a* ***int*** *inizializzato col valore di questo indirizzo di memoria e lo restituirò nel punto del programma in cui è stata richiesta la mia opera. Se non dovessi riuscire a trovare un'area di memoria adeguata alle richieste il puntatore che genererò sarà un puntatore NULLO."*   
  
Questo dovrebbe bastare a capire che nella riga *// 5* si effettua un'inizializzazione di *k* a partire da un puntatore inizializzato, come nella riga *// 4*; solo che stavolta l'area di memoria indirizzata è NUOVA e l'unica variabile di tipo *int* che può legittimamente contenere, e reperibile, come sempre, attraverso *k[0]*, NON È STATA INIZIALIZZATA A SUA VOLTA, come invece accade nelle righe *// 7* e *// 8*.  
Il successivo *cout*, pertanto, fa apparire qualcosa di aleatorio e di potenzialmente diverso da un calcolatore all'altro: situazione da EVITARE. Le righe *// 7* e *// 8* mostrano come si possa eseguire l'immediata inizializzazione della variabile puntata dal puntatore, contestualmente con l'inizializzazione di quest'ultimo, e si vede che non è necessario usare una costante, potendosi utilizzare qualsiasi espressione valutabile di tipo adeguato.  
Naturalmente l'inizializzazione della variabile puntata può tranquillamente essere procrastinata: basta che prima o poi la si faccia e comunque PRIMA di utilizzare *k[0]* come operando destro di qualsiasi operatore che non sia *>>* con *cin* come operando sinistro.   
  
Nella riga *// 6*, infine, si vede come l'operatore *new* possa catturare nuova memoria in misura maggiore di quella necessaria per una singola variabile, usando perfino il valore di un'espressione valutabile, **NON** solo una costante, e come si possano immediatamente inizializzare tutte le variabili ordinatamente puntate generando sostanzialmente l'equivalente di un *array*.  
Si osservi che per l'operatore *new*, diversamente da quanto accade con gli *array*, non sarebbe stato lecito lasciar vuote le parentesi quadre e che l'immediata inizializzazione è pur sempre facoltativa: se la si esegue, come nel caso presente, il compilatore si spiace solo di non poter essere sicuro che la lista degli inizializzatori non sia troppo lunga e mette in guardia il programmatore del suo non essere in grado di valutare l'ammontare della memoria catturata in relazione al numero di inizializzatori inseriti; ma dopo questo finisce col fidarsi dell'autore del codice avendogli/le previamente scaricata addosso ogni responsabilità per eventuali accessi illegittimi alla memoria.   
La linea commentata non è ERRATA sul piano della sintassi, ma su quello della legalità dell'accesso (i/3 fa 2!); se si prova a togliere il commento il programma si compilerà ugualmente e ugualmente funzionerà nel 90% delle esecuzioni, ovviamente scrivendo fandonie al posto di *k[2]*, ma quando incapperete nel restante 10%, e il programma si interromperà, prendetevela con VOI STESSI, **NON** col compilatore e men che meno con me.   
  
Come esistono *array* contenenti altri *array*, così esistono puntatori che puntano altri puntatori, fino a qualsiasi livello di ricorsività. Ad esempio

int   \*\* x;

è la dichiarazione di un puntatore *x* la cui area di memoria indirizzata è piena di altri puntatori a *int* piuttosto che di semplici *int*, mentre

int   \* \* \* y;

è la dichiarazione di un puntatore *y* la cui area di memoria indirizzata è piena di oggetti della stessa natura del precedente *x*, e via andando con l'aumentare del numero di asterischi nella dichiarazione, fino a oltre ogni gestibilità da parte dell'uomo, prima ancora che del calcolatore. Non dovrebbe esserci nessuno che non veda la potenza di tutto questo...  
L'inizializzazione di simili puntatori non presenta grandi novità: richiede SOLO coerenza e va fatta utilizzando correttamente l'operatore *new*: ad esempio i nostri *int \*\*x* e *int \*\*\*y* potrebbero essere lecitamente inizializzati così:

x = new   int   \*  [2];  
x[0] = new   int   [3];  
x[1] = new   int   [4];  
y = & x;

Si noti che *x* è ben diverso da un *array* che contenga altri *array*, perché in quel caso gli *array* interni hanno uguale capienza, e che *y* diventa invece equivalente a un *array* contenente UN SOLO ELEMENTO che è *x*.  
Naturalmente *y* si poteva anche inizializzare a partire da memoria **nuova** nel seguente modo:

y = new   int   \*  \*[2];  
y[0] = new   int   \*[3];  
y[1] = new   int   \*[4];  
y[0][0] = new   int   [5];  
// altre due analoghe per y[0][1] e y[0][2]   
y[1][0] = new   int   [2];  
// altre tre analoghe per y[1][1], y[1][2] e y[1][3]

Per concludere questo canto si introdurrà l'UNICA MANIERA VALIDA E SICURA di creare *array* dinamicamente inizializzati senza dover ricorrere all'uso della *Standard Template Library*, che è assolutamente prematuro per il vostro livello di conoscenza.  
Si ricorda che un *array* è inizializzato dinamicamente quando la sua capienza non è una costante il cui valore sia noto all'atto della compilazione, ma piuttosto una variabile intera di cui si apprenda il valore solo quando il programma passa alla fase esecutiva. Nel canto precedente si è detto che taluni compilatori ammettono una sequenza di codice come la seguente:

int n;  
cout << "fornisci il valore di n ", cin >> n;  
int x[n];

ma è assolutamente preferibile e meglio garantito sostituirla con questa:

int n;  
cout << "fornisci il valore di n ", cin >> n;  
auto x = new int[n];

che ha l'ulteriore pregio di poter essere immediatamente generalizzata ad *array* *multi-indicizzati* come qui appresso:

// dichiarazione e inizializzazione  
// di un *array* dinamico multi-indice  
int n;  
cout << "fornisci il valore di n ", cin >> n;  
double (\*z)[3][6] = new double[n][3][6];

L'uso della parola *auto*, appartenente al vocabolario del linguaggio, istruisce il compilatore a desumere il tipo dell'*array* dichiarando da quello trasmesso all'operatore *new*; si osservi che SOLO la prima, eventualmente unica, capienza può essere dinamicamente assegnata e che le eventuali capienze successive devono corrispondersi ESATTAMENTE. Si tratta, in ultima analisi, di qualcosa di simile all'inizializzazione effettuata tramite inizializzatori racchiusi tra graffe, MA SI OSSERVI CHE, NEL CASO ATTUALE, NON C'È INIZIALIZZAZIONE DEL CONTENUTO DELL'*ARRAY*.  
  
Un ultimo paragrafo di questo canto viene scritto per parlare dei puntatori a *void*: resistete alla tentazione del fallace processo induttivo secondo cui, se un puntatore a *int* è un indirizzatore di *int*, allora un puntatore a *void* indirizza dei *void*, perché questa frase è priva di significato, non avendolo i *void* stessi quando non siano funzioni; e sui puntatori a funzione, che ESISTONO, si tornerà al tempo opportuno. Per giunta un puntatore a *void* è PRIVO di algebra di puntamento, nel senso che, se *v* è un tale puntatore, *v+1* non è NULLA, se non un ERRORE, diversamente da quanto accade per i puntatori di ogni altro tipo.  
Un puntatore a *void* indirizza pertanto un'area di memoria completamente amorfa e al cui interno esso non è capace di spostarsi in alcun modo, perché non ha alcuna idea di quanto spostarsi: NON ESISTE *sizeof void*!.  
Il suo limite, appena descritto, è la sua stessa utilità: non avendo forma, l'area di memoria puntata da un puntatore a *void* può assumere, di volta in volta, quella che più conviene, con l'ausilio dell'operatore di *casting* che si introdurrà di qui a poco, un po' come una massa liquida assume la forma del recipiente.  
Considerate il seguente pseudocodice:

void \* v; // dichiarazione di un puntatore a *void*  
v = new char[24]; // inizializzazione di v  
/\*N.B. il tipo serve SOLO all'operatore *new* per stabilire  
QUANTA memoria allocare: in questo caso v punta 24 *bytes*.  
Si sarebbe potuto sostituire *char* con qualsiasi altro tipo,  
TRANNE PROPRIO *void*; ad esempio usando *double*  
e lasciando il 24 si sarebbero indirizzati 192 *bytes*.\*/  
char \*c = (char \*)v;  
// dichiarazione di un puntatore a *char*  
// inizializzato con *v*; in questo modo si   
// dà alla memoria puntata la forma che avrebbe  
// se puntasse dei *char*  
// ..... molte, molte linee che sfruttano QUELLA memoria  
int \*i = reinterpret\_cast<int \*>(v);  
// ora la STESSA memoria, senza bisogno  
// di sprecar tempo  
// a rilasciarla e riallocarla, può essere usata  
// per contenere 6 *int* che saranno indirizzati da *i*  
// ..... e via dicendo

Nelle dichiarazioni di *c* e di *i* viene per l'appunto applicata al puntatore *v* l'operazione di *casting*, che vedremo ben presto più approfonditamente, in due forme tra loro equivalenti. A tutti i fini pratici è come se *c* e *i* fossero inizializzati rispettivamente con *new char[24/sizeof(char)]* e *new int[24/sizeof (int)]*, col vantaggio che si ricicla sempre la stessa memoria piuttosto che consumarne di nuova.

**Canto decimo: i cicli**

Riconsiderate questo segmento di codice, proveniente dal precedente canto,

int    un\_array\_utile[3];  
cout   
<< (un\_array\_utile[0] = 10)   
<< " si trova in "   
<<un\_array\_utile << '\n'  
<< (un\_array\_utile[1] = 20)   
<< " si trova in "  
<<un\_array\_utile+1 << '\n'  
<< (un\_array\_utile[2] = 30)   
<< " si trova in "  
<<un\_array\_utile+2   
<<'\n';

e immaginate, per un istante, di aver inserito tra le parentesi quadre della dichiarazione di *un\_array\_utile*, al posto di 3, 8421. Occorrerà dunque scrivere 25263 linee di programma solo per inizializzare e fare scrivere su *output* ciascun elemento contenuto nell'*array* e il suo indirizzo di memoria? E se 8421 diventasse 35000?

**Ecco perché servono i cicli!**

Immaginate di sostituire il codice precedente con quello che segue:

int   i = -1, un\_array\_utile[3];  
while(++i < 3)   
cout  
<< (un\_array\_utile[i] = 10\*(i+1))   
<< " si trova in "   
<<un\_array\_utile + i << '\n';

Se farete eseguire entrambi i codici (FATELO, DUNQUE!) vedrete che, a parte la fisiologica non coincidenza dei valori degli indirizzi in memoria, peraltro già incontrata e discussa, produrranno un risultato identico, col vantaggio che, nel secondo codice, un eventuale aumento di capienza dell'*array* richiederebbe **solo** sostituire coerentemente il valore contenuto all'interno delle parentesi che seguono la parola *while* per ottenere che l'output si estenda quanto necessario.  
Per giunta, se al posto di un *array* si usasse un puntatore, la seguente variante:

int   i = -1, n, \*stavolta\_uso\_un\_puntatore;  
cout << "digita un intero positivo QUALUNQUE ",  
cin >> n,  
stavolta\_uso\_un\_puntatore = new int[n];  
while(++i < n)   
cout  
<< (stavolta\_uso\_un\_puntatore[i] = 10\*(i+1))   
<< " si trova in "   
<<stavolta\_uso\_un\_puntatore + i << '\n';

non richiederebbe NEPPURE QUANTO APPENA DETTO, a riprova dell'assioma secondo cui un programma è tanto meglio scritto quanto minore è il numero di costanti esplicite che contiene, e del fatto che i puntatori sono più potenti degli *array*.  
  
La parola di vocabolario *while* introduce la più semplice forma di ciclo prevista dal linguaggio: un ciclo non è altro che un ben delimitato pezzo di programma che il programmatore decide di far eseguire RIPETUTAMENTE, **fino a quando gli/le CONVENGA**.  
  
Nel caso attuale il pezzo di programma che costituisce il ciclo è **SOLO L'ESPRESSIONE** che inizia con *cout*: l'unica espressione presente; **in generale** l'estensione del ciclo si misura:

1. dalla stessa parola *while*, che lo inizia, ivi compreso il contenuto delle parentesi tonde, richieste dalla sintassi del linguaggio;
2. fino a, in alternativa:
   * il **PRIMO** punto e virgola incontrato nel codice se dopo *while* non si apre volontariamente un ambito: si parla di ciclo costituito da una sola espressione;
   * l'**INTERO** ambito eventualmente, e volontariamente, aperto **subito dopo** la clausola *while(  )*, e in questo caso il ciclo potrebbe estendersi per migliaia di espressioni e linee di codice.

**Ci si cacci perennemente in testa** questa topologia del ciclo, perché è la stessa anche per tutte le altre forme di ciclo contemplate dal linguaggio: ci sarà SOLO da sostituire *while* con qualcos'altro.  
  
Analizziamo ora più dettagliatamente l'ultimo codice riportato sopra; supponendo che quando il programma fa uscire la richiesta di un numero intero positivo qualunque, il *pan troglodytes* (scimpanzé) seduto alla tastiera risponda coerentemente, la prima cosa che vien fatta è l'appropriata inizializzazione del puntatore tramite l'operatore *new*.  
Successivamente inizia il ciclo e il programma, per prima cosa, compie la valutazione dell'espressione contenuta **OBBLIGATORIAMENTE** entro le parentesi a destra della parola *while*; si tratta di un'unica espressione e NON DEVE ESSERE terminata col punto e virgola: è questa una delle poche eccezioni alla regola secondo cui un'espressione termina quando si incontra tale carattere. In tale espressione si confronta il risultato della sottoespressione *++i* col valore della variabile digitata dallo scimpanzé e dato che *i* era stata inizializzata con -1 e che l'operatore ++ incrementa di 1 il valore corrente (IMPARATELO!), si finisce per confrontare 0 con *n*: se lo scimpanzé si è comportato bene, come ipotizzato, tale confronto dà esito positivo, ossia VERO, e questo costituisce semaforo verde per l'effettiva esecuzione del ciclo, ossia l'inizializzazione del primo elemento puntato dal puntatore (*i* VALE ZERO!) e la sua scrittura.  
A questo punto, essendo completata l'esecuzione del ciclo, il programma TORNA A VALUTARE l'espressione *++i < n*, nella quale l'operatore ++ agisce novamente, portando il valore di *i* a 1; se 1 sia o no minore di *n* dipende da che cosa ha digitato lo scimpanzé: se avesse digitato appunto 1 l'espressione sarebbe già falsa e scatterebbe semaforo rosso alla reiterazione del ciclo, facendo proseguire il programma dall'istruzione immediatamente successiva al ciclo stesso.  
  
La morale è chiara: il ciclo viene ripetuto **ESATTAMENTE** *n* volte, non una di più né una di meno, ossia **ESATTAMENTE TANTE VOLTE QUANTI SONO** gli elementi puntati dal puntatore: a ogni ripetizione del ciclo viene inizializzato e scritto un elemento che non era ancora stato inizializzato, di modo che, all'uscita dal ciclo, **TUTTI GLI ELEMENTI PUNTATI, E SOLO GLI ELEMENTI PUNTATI** sono stati inizializzati, **QUALUNQUE FOSSE IL LORO NUMERO**.  
  
Non havvi chi non veda, spero, la portata di tutto questo.  
  
Naturalmente tutte le varianti plausibili concernenti l'inizializzazione di *i* sono ugualmente ammissibili, purché sia fatta salva la coerenza degli effetti del ciclo; per esempio, inizializzando *i* con 0 (zero) invece che con -1, il ciclo si sarebbe dovuto modificare in questo modo:

while(++i <= n)   
cout  
<< (stavolta\_uso\_un\_puntatore[i-1] = 10\*i))   
<< " si trova in "   
<<stavolta\_uso\_un\_puntatore + i - 1<< '\n';

si apprezzino (E SI CAPISCANO!) le variazioni apportate al codice per conseguire lo STESSO risultato. In particolare si cominci a prendere dimestichezza con **ogni nuovo operatore che incontrate**: qui, ad esempio, gli operatori di confronto *<* e *<=* e quello di incremento unitario *++*.  
  
A proposito dell'operatore di incremento vanno dette due cose: la prima è che c'è anche quello di decremento, *--*, e la seconda è che entrambi esistono anche in forma postfissa, o suffissa, vale a dire scritti come *i++* o *i--* invece di *++i* o *--i*; la sostanziale differenza tra le due forme è il momento in cui gli operatori agiscono nell'espressione in cui compaiono: nella forma prefissa agiscono per primi, mentre nella forma postfissa agiscono per ultimi. Pertanto, mantenendo l'inizializzazione di *i* con 0 (zero), il codice che produce gli stessi effetti, ma usando la forma suffissa dell'operatore, è il seguente:

// la **prima volta** è **0** a confrontarsi con n  
while(i++ < n)  
cout  
<< (stavolta\_uso\_un\_puntatore[i-1] = 10\*i))   
<< " si trova in "   
<<stavolta\_uso\_un\_puntatore + i - 1<< '\n';

mentre, se si tornasse a inizializzare *i* con -1, diverrebbe:

// la **prima volta** è **-1** a confrontarsi con n-1  
while(i++ < n-1)  
cout  
<< (stavolta\_uso\_un\_puntatore[i] = 10\*(i+1)))   
<< " si trova in "   
<<stavolta\_uso\_un\_puntatore + i << '\n';

Tutto è talmente ovvio che se ne lascia la verifica (E LA COMPRENSIONE!) a voi. Si osservi solo che i quattro cicli proposti, come è coerente e giusto che sia, sono tutti diversi tra loro e proprio per questo producono il medesimo effetto.  
  
L'espressione che compare all'interno delle parentesi di *while*, in realtà, può anche essere una dichiarazione di una **singola** variabile, accompagnata da **immediata inizializzazione** e ancora NON terminata col punto e virgola; in tal caso la variabile così dichiarata ha come proprio ambito esclusivamente il ciclo stesso e non è quindi utilizzabile al di fuori. Si consideri il seguente esempio:

int a[ ]{1, 3, 7, 0, 5, 6}, i=0;  
while(int c = a[i++]) cout << c << '\n';  
// cout << c << '\n'; // ERRORE

Questo segmento di programma scrive, uno per riga, i valori 1, 3 e 7.  
FATE ESEGUIRE IL CODICE e provate a capire quello che succede, PRIMA di leggere la spiegazione qui appresso.  
  
Quando si giunge all'inizio del ciclo, il programma inizializza la variabile *c* col valore del **primo** elemento di *a* (*i* VALE ZERO), **dopo di che** incrementa *i* di UNO; il valore assunto da *c*, il primo elemento di *a*, è 1, che è valutato come equivalente booleano di *true*; TUTTO È VERO QUEL CHE NON È ZERO: quindi il ciclo si esegue e il valore di *c* viene scritto. Il ciclo si esaurisce con questa scrittura perché c'è il punto e virgola, e quindi si torna a valutare la dichiarazione in *while*. Questa volta *c* è inizializzato col valore del **secondo elemento** di *a* (*i* VALE UNO, e solo DOPO che l'inizializzazione di *c* ha avuto luogo si incrementa). Anche il secondo valore contenuto in *a*, e acquisito da *c*, è booleanamente VERO, e quindi viene scritto perché il ciclo viene rieseguito. La cosa va avanti fino a quando *c* non viene inizializzato con il valore del quarto elemento di *a*, che è ZERO e quindi booleanamente FALSO: il ciclo NON VIENE ESEGUITO, zero non viene scritto, e il programma prosegue dal codice immediatamente successivo al punto e virgola che chiude il ciclo stesso.  
Tale codice **NON PUÒ ESSERE** la linea opportunamente commentata, perché lì *c* **NON È DICHIARATO**: provate a eseguire togliendo il commento e vedrete.   
  
Oltre alla forma di ciclo fin qui ampiamente discussa, il linguaggio ne offre altre TRE, tutte, come già detto, con la stessa topologia; la più simile alla prima citata è quella introdotta dalla parola di vocabolario *do*, e del resto continua a utilizzare anche la parola *while*. La sua sintassi è infatti

do  
// ciclo  
while(......);

ove, OVVIAMENTE, al posto dei puntini va qualcosa di analogo a quanto già visto e, altrettanto ovviamente, il commento *// ciclo* funge da segnaposto per il contenuto effettivo del ciclo stesso, secondo le due diverse realizzazioni possibili della propria topologia. Il punto e virgola a destra della clausola *while* è OBBLIGATORIO. Si considerino i seguenti due esempi a confronto:

**Esempio 1:**  
  
int c;  
cin >> c;   
while(c)  
cout << "NEL ciclo " << c << '\n',  
cin >> c;   
cout << "FUORI DAL ciclo " << c << '\n';  
  
**Esempio 2:**  
  
int c;  
cin >> c;   
do  
cout << "NEL ciclo " << c << '\n',  
cin >> c;   
while(c);  
cout << "FUORI DAL ciclo " << c << '\n';

I due esempi differiscono SOLO per la forma di ciclo utilizzata; tutto il resto coincide, compresa l'espressione usata all'interno delle parentesi in *while*. Se venissero digitati da tastiera gli stessi valori in entrambi i casi, quale sarebbe la differenza?  
  
ESEGUITE I DUE CODICI E DIGITATE LA **STESSA** SEQUENZA DI NUMERI INTERI, FACENDO IN MODO CHE VI SIANO ANCHE DUE ZERI CONSECUTIVI.  
  
**Vista la differenza?**   
  
**NO?**   
  
Consiste nella cronologia: nel ciclo introdotto da *do* la decisione sulla legittimità dell'esecuzione del ciclo viene presa **SOLO DOPO** che il ciclo **È STATO ESEGUITO**. Ne segue che con *do* il ciclo viene sempre eseguito **ALMENO UNA VOLTA**, mentre nella prima forma il ciclo potrebbe anche essere **ESEGUITO NEPPURE UNA SOLA VOLTA**. Provate a eseguire i due codici digitando SUBITO lo 0, se non l'avete già fatto, e vedrete.  
  
La terza e la quarta delle quattro forme di ciclo usano entrambe la parola di vocabolario *for*, ma in modo totalmente diverso per quanto riguarda la sintassi e il significato. Cominciamo dalla terza forma di ciclo, la prima che usa *for*; la sintassi è:

for( elemento1 ; elemento2 ; elemento3 )  
// ciclo

ove *// ciclo* indica, come prima, una delle due possibili topologie del ciclo stesso e ciascuno dei tre *elementi* PUÒ ESSERE OMESSO, diversamente da quanto avviene con *while* cui non si può lasciar vuota la coppia di parentesi tonde. Ed è **proprio per questo** che, in questo caso, la sintassi **ESIGE LA PRESENZA** di DUE, e NON PIÙ DI DUE, segni di punto e virgola all'interno delle parentesi tonde OBBLIGATORIE.  
  
*elemento1*, quando c'è, può essere un'espressione qualsiasi, valutabile o no, OPPURE una **singola dichiarazione** di una o più variabili immediatamente inizializzate; in questo caso, come per *while*, la visibilità della/e variabile/i così dichiarata/e è circoscritta al ciclo stesso e non si estende al di fuori.  
Se *elemento1* è presente viene valutato/eseguito **UNA VOLTA SOLA**, appena l'esecuzione del programma giunge a *for*, e poi **MAI PIÙ**; se non è presente...il problema non si pone.  
  
*elemento2*, quando c'è, è **morfologicamente** e **funzionalmente IDENTICO** a quanto descritto essere all'interno delle parentesi di *while*, quindi non si spenderanno altre parole in proposito; quando non c'è è interpretato come la costante booleana *true* (VERO!).  
  
*elemento3*, quando c'è, può essere un'espressione qualsiasi, valutabile o no, come accade a *elemento1*, ma NON a *elemento2* che NON PUÒ ESSERE NON VALUTABILE, ma NON UNA DICHIARAZIONE, come invece può avvenire tanto a *elemento1* quanto a *elemento2*. Se è presente viene valutato/eseguito **ALLA FINE DI OGNI ESECUZIONE** del ciclo, IMMEDIATAMENTE PRIMA che si rinnovi la valutazione/esecuzione di *elemento2* per controllare se il ciclo debba essere eseguito ancora; se non è presente...il problema non si pone.  
  
In definitiva, il precedente codice formale parrebbe del tutto equivalente, per quanto fin qui scritto, al seguente codice, altrettanto formale:

elemento1;  
while(elemento2)   
{   
// ciclo  
elemento3;  
}

e, in effetti, È EQUIVALENTE, in una buona percentuale di situazioni...peccato che esistano le altre che rendono i due codici formali NIENTE AFFATTO EQUIVALENTI. Pensandoci per un nanosecondo, se ci fosse un 100% di equivalenza nessun essere ragionevole potrebbe comprendere né la necessità né l'utilità di due costrutti distinti: chiamiamoli rispettivamente **costrutto *for*** e **costrutto *while***. Le differenze SOSTANZIALI sono le seguenti, in ordine crescente di rilevanza:

* se *elemento2* è vuoto il costrutto *for* è valido, mentre il costrutto *while* genera errore; questo era già stato detto, e per venirne a capo basterebbe che il linguaggio ammettesse parentesi vuote nel costrutto *while* o vietasse l'omissione di *elemento2*, ma perché trattare *elemento2* in maniera diversa dai suoi fratelli, generando solo confusione?
* se *elemento3* NON è vuoto e in *// ciclo* si incontrasse l'istruzione *continue*, di cui si parlerà più diffusamente in un prossimo canto, *elemento3*, nel costrutto *while*, NON SAREBBE ESEGUITO; nel costrutto *for*, invece, **SÌ**;
* se *elemento1* NON è vuoto, ed è una dichiarazione, nel costrutto *while* tale dichiarazione si estenderebbe **ALL'ESTERNO DEL CICLO**; nel costrutto *for*, invece, **NO**.

Tutto ciò premesso e dibattuto ecco di seguito un segmento di codice autentico in cui si usa *for* per inizializzare tramite lettura da tastiera, in maniera didatticamente un po' cervellotica, giusto per sbizzarrirsi, tutti gli elementi di due *array* di uguale capienza, sempre avendo a disposizione una bertuccia intelligente:

int a[5], b[5];  
for(int i = 0, j = 0; i < 5; ++i, j += 2)  
cout << " valore degli interi numero "   
<< i+1 << " per a e b? ",   
cin >> a[i] >> b[j/2];

Si osservi che ogni elemento contenuto in *for* aderisce a quanto detto sopra e si ricostruisca mentalmente il flusso dell'esecuzione. L'operatore *+=* incrementa il proprio operando sinistro col valore del proprio operando destro (IMPARATELO!), ed esistono anche, in piena coerenza, gli analoghi operatori *-=*,    *\*=*,    */=*.  
  
L'ultima forma di ciclo, denominata in anglo-americano *range for*, e introdotta per la prima volta nello standard 2011 del linguaggio, usa la parola *for* con la sintassi seguente:

for(elemento1 : elemento2)  
// ciclo

Fermo restando il consueto significato del finto commento, questa volta non si può omettere un bel nulla e i due *elemento* devono essere separati con il carattere ':' (due punti).  
  
Allo stato attuale delle vostre conoscenze *elemento2* può essere SOLO il nome di un *array* dichiarato; quando crescerete vi verrà detto che può ANCHE essere ben altro, e ben di più.  
  
Quanto a *elemento1*, DEVE essere la dichiarazione di una variabile, la cui visibilità, come per gli altri cicli, resterà CONFINATA entro il ciclo stesso, OPPURE DI UN RIFERIMENTO, di tipo compatibile con quello di *elemento2*: lo stesso tipo, OPPURE un tipo desunto dal compilatore stesso mercé l'uso della parola di vocabolario *auto*.  
  
*// ciclo* viene eseguito **ESATTAMENTE TANTE VOLTE** quanti sono gli elementi contenuti in *elemento2*, e, in ognuna di tali esecuzioni, la variabile dichiarata in *elemento1* assume ORDINATAMENTE il valore del contenuto di *elemento2* corrispondente a tale esecuzione, OVVERO, secondo che cosa è stato dichiarato, ACCEDE ORDINATAMENTE alla sua locazione in memoria.  
  
È per questa ragione che *elemento2* NON PUÒ ESSERE, ad esempio, il nome di un puntatore (verrebbe generato un ERRORE) dato che, vista la variabilità della memoria indirizzabile tramite puntatori, il compilatore non potrebbe sapere quante volte far eseguire il ciclo.  
  
Nei seguenti segmenti di codice (FATELI ESEGUIRE!) si esemplifica questo tipo di ciclo in alcuni contesti, simili tra loro, ma del tutto diversi, un po' come le vignette *trovate le differenze* nelle riviste di enigmistica:

**Segmento 1:**  
  
int a[ ] {1, 2, 3};  
for(int i:a) cout << i << '\n';  
  
**Segmento 2:**  
  
int a[ ] = {1, 2, 3};  
for(auto i:a) cin >> i;  
// digitate tre valori interi diversi da 1, 2, 3  
// terminando ciascuno con "Invio"  
for(int &i:a) cout << i << '\n';  
  
**Segmento 3:**  
  
int a[ ] = {1, 2, 3};  
for(auto &i:a) cin >> i;  
// digitate tre valori interi diversi da 1, 2, 3  
// terminando ciascuno con "Invio"  
for(int i:a) cout << i << '\n';  
  
**Segmento 4:**  
  
int b[ ][2]{1, 2, 3, 4};  
for(auto &i:b) cout << i << '\n';  
  
**Segmento 5:**  
  
int b[ ][2]{1, 2, 3, 4};  
for(auto &i:b) cout << \*i << '\n';  
  
**Segmento 6:**  
  
int b[ ][2]{1, 2, 3, 4};  
for(auto i:b) cout << \*i << '\n';  
  
**Segmento 7:**  
  
int b[ ][2]{1, 2, 3, 4};  
for(auto i:b) cout << i[1] << '\n';

**LI AVETE ESEGUITI?**  
  
**NOOO???**  
  
**ESEGUITELI, HO DETTO!**  
  
Avete capito che cosa succede? Provate a spiegarvelo da soli, **PRIMA** di leggere la spiegazione qui sotto, se volete essere persone serie, rileggendo bene quello che è stato detto sopra su questo tipo di ciclo, e rileggendolo **CON TUTTI I VOSTRI NEURONI ACCESI E FOCALIZZATI SU QUELLO CHE LEGGETE**, centellinando con la mente **OGNI SINGOLA PAROLA:** è così che si impara!  
  
Nei **Segmenti 1, 2, 3** si dichiara sempre lo stesso *array* (*a*), contenente gli stessi TRE interi: 1, 2, 3, nell'ordine; il segno = nell'inizializzazione, si sa, è facoltativo. Nel primo segmento, *for* dichiara una semplice variabile *i*, di tipo IDENTICO a quello dei contenuti di *a*, che, come spiegato, in ognuna delle TRE esecuzioni del ciclo **assumerà ordinatamente** il valore del corrispondente contenuto di *a*: non dovrebbe quindi essere alcuna sorpresa il fatto che si vedano apparire scritti, uno per riga, tali valori.  
La frase *assumerà ordinatamente il valore*, se uno centellina le parole sorseggiandole a una a una con la mente, significa che, se *i* assume il valore vuol dire che, prima di assumerlo, non l'aveva; in altre parole significa (è la LOGICA che lo dice, ABITUATEVI!) che, a ogni esecuzione del ciclo, *i* viene reinizializzato tacitamente da parte del calcolatore, con l'esecuzione implicita dell'operatore di assegnamento in un'espressione del tipo *i = a[  ]*, con all'interno delle quadre un contatore, altrettanto implicito, che va da 0 alla *capienza\_array - 1*. Tutto questo AVVIENE, anche se non si vede, e, in definitiva, ciò è dovuto al fatto che *i*, essendo una variabile come tutte le altre, HA LA SUA celletta di memoria, NIENTE AFFATTO INDIRIZZATA da *a*, e i valori degli elementi di quest'ultimo, per tener fede al significato del ciclo, VI DEVON ESSERE RICOPIATI, di volta in volta.  
  
Forti della digestione di quanto appena asciolto, si dovrebbe capire immantinente che l'inizializzazione per lettura tramite l'oggetto *cin*, compiuta nel **Segmento 2**, **è completamente sbagliata**, e non sul piano sintattico, ma sul piano del buon funzionamento del programma. Infatti il compilatore non si lagna punto, anche perché la dichiarazione fatta con *auto* viene facilmente riconosciuta come del tutto equivalente ad averla fatta con *int*.   
Ragionando col cervello acceso si DEVE desumere che l'operazione di lettura inizializza *i* nella SUA celletta di memoria che, abbiamo detto, non ha niente da spartire con le cellette indirizzate da *a*, i cui valori iniziali, non per nulla, riappaiono intonsi all'esecuzione del secondo *for*; a NULLA serve che *i*, prima di essere inizializzata con *cin*, fosse già stata inizializzata in precedenza con uno dei valori di *a*, dato che questa inizializzazione va irrimediabilmente perduta per sovrascrittura da parte dell'inizializzazione immediatamente successiva.  
E poiché all'estinzione del ciclo la variabile *i* stessa svanisce nel nulla, anche tutte le digitazioni sulla tastiera se ne vanno a schifìo, come si dice in Sicilia, e perfino il *macaca sylvanus* (bertuccia di Gibilterra) deputato a compierle avrebbe, per questa volta, il diritto di esternare al programmatore la propria riprovazione (mi sembra di vederlo...).  
Nello stesso **Segmento 2** il secondo *for* dichiara novamente *i*. Si tratta di un'ALTRA *i*, quella precedente è **MORTA**; ma stavolta non come semplice variabile bensì come riferimento a un intero. È questo il significato del segno & preposto al nome in una dichiarazione; ma, se ricordate bene, lo stesso segno, usato in un'espressione, diventa per ciò stesso un operatore che produce come risultato l'indirizzo nella memoria dell'operando cui si applica. Andate a riguardare come si può inizializzare un puntatore.  
Ne segue che, stavolta, a ogni esecuzione del ciclo, *i* **non assume ordinatamente il valore** di ciascun elemento di *a*, bensì, come detto sopra, **accede ordinatamente alla locazione in memoria** di tali elementi, il che significa che, a ogni esecuzione, *i* **È** uno di tali elementi, senza che avvenga alcuna ricopiatura di valori: in altre parole *i* diventa un semplice *alias* per quello che avevamo indicato come *a[ ]*, con all'interno delle quadre il famigerato contatore implicito, ed è **QUESTO** quello che scrive *cout*, ossia il valore di *a[ ]* IN PERSONA e NON QUELLO DI UNA VARIABILE DIVERSA che ha assunto quel valore.  
  
L'auspicabile comprensione di questa verità fondamentale (**SCOLPITELA NEL CERVELLO,** dovesse volerci lo scalpello) dovrebbe bastare a rendervi ragione del corretto funzionamento del **Segmento 3**.  
  
Nei quattro **Segmenti** finali si usa un *array* (*b*) con due capienze. Nulla cambia, perché ciò che si dichiara nei diversi *for* si deve tuttora riferire, in un modo o nell'altro, ai **contenuti di *b***. E quali sono i contenuti di *b*? Appare OVVIO che si tratta dei DUE *array* (*b[0]* e *b[1]*), ciascuno dei quali contiene le coppie di interi {1,2} e {3,4} rispettivamente, e che quindi TUTTI i cicli in ciascuno dei quattro esempi saranno eseguiti DUE VOLTE. Riguardatevi il canto sugli *array*, se non ve li ricordate, MA DOVRESTE COMINCIARE A IMPARARE CHE **TUTTO** VA RICORDATO.  
Ciò che *i* sarà in ciascun esempio dipende da come è stato dichiarato; se con il segno & si tratterà dei due *array* IN PERSONA: *b[0]* alla prima esecuzione e *b[1]* alla seconda; se invece sarà stato dichiarato SENZA il segno & si tratterà, di esecuzione in esecuzione, di UN ALTRO *ARRAY*, di capienza 2, in cui saranno ricopiati i valori contenuti nei due *b[ ]*. Si osservi l'assoluta utilità, nel presente contesto, della parola *auto* nell'effettuare la dichiarazione.  
Resta solo da dare un significato al segno di asterisco nelle espressioni *\*i* che compaiono nei **Segmenti 5,6**: quando appare in un'espressione, come operatore unario, ossia con un solo operando, l'asterisco viene denominato *operatore di dereferenza*; il suo UNICO OPERANDO **DEVE ESSERE UN PUNTATORE** e, nel presente contesto, *i* è tale, perché un *array* **È** un puntatore, e la sua azione consiste nel ricavare il valore dell'oggetto che si trova all'indirizzo rappresentato da quel puntatore: per farla breve *\*i*, in un'espressione, EQUIVALE a *i[0]*, solo che richiede due caratteri in meno da digitare per scriverlo.  
  
Tutto ciò spiegato e scritto, il comportamento di ognuno degli ultimi quattro esempi dovrebbe risultare trasparente. O no?  
  
Concluderemo questo canto con un altro assioma:  
  
**NON BISOGNA MAI AVER PAURA DEI CICLI, NEPPURE DEI CICLI ETERNI**, che, anzi, talvolta sono la soluzione più efficace a determinati problemi, che vedremo, **MA DA QUESTI OCCORRE SEMPRE ALLESTIRE UNA VIA D'USCITA**.

**Seconda pausa di riflessione**

Dopo aver riletto con attenzione tutti i canti fin qui proposti (dal canto 0 al canto 10), provate a scrivere da voi e a eseguire un programma, costituito dalla SOLA funzione *main*, che faccia le seguenti cose:

* dichiari due *array* di uguale capienza e un puntatore, tutti dello stesso tipo;
* dichiari, in aggiunta, UNA SOLA variabile intera con visibilità estesa a tutta la funzione *main*;
* inizializzi il puntatore in maniera tale da indirizzare il doppio della capienza di ciascuno dei due *array*;
* usi un ciclo *for* (della prima specie) per inizializzare il contenuto del primo *array* con valori opportuni letti da tastiera;
* usi un ciclo *for* (della seconda specie) per inizializzare ordinatamente il contenuto del secondo *array* con valori opposti a quelli del contenuto del primo;
* usi un ciclo *do* per inizializzare ordinatamente la prima metà dei valori puntati dal puntatore con le somme dei contenuti dei due *array*;
* usi un ciclo *while* per inizializzare ordinatamente la seconda metà dei valori puntati dal puntatore con le differenze dei contenuti dei due *array*;
* usi il ciclo che ritenete più idoneo (ma UNO SOLO) per scrivere ordinatamente sul canale di *output* i contenuti dei due *array* e quelli puntati dal puntatore;

Quel che si dovrà vedere saranno quaterne di due numeri tra loro opposti seguiti da zero e dal doppio del primo numero, per ogni valore digitato da tastiera, che dovrà coincidere (come è evidente) col primo numero di ogni quaterna.   
  
**FATELO SUL SERIO, SE VOLETE IMPARARE**, altrimenti sarà peggio per voi.

**Canto undicesimo: istruzioni condizionate, cicli eterni e vie d'uscita**

C'è chi sostiene che ogni linguaggio di programmazione non è altro che una successione ordinata di cicli e di istruzioni condizionate, ossia tali per cui l'esecuzione sia sottoposta al verificarsi di qualche adeguato evento.  
Benché una simile generalizzazione sia alquanto riduttiva, il C++ non se ne sottrae e, come ogni altro linguaggio, offre le proprie istruzioni condizionate, la prima delle quali ha la sintassi:

if( espressione ) ........... //segmento di programma

in cui *if* è la parola di vocabolario da usare, la coppia di parentesi È OBBLIGATORIA, e al posto dei puntini va OVVIAMENTE il segmento di programma citato dal commento. Tale segmento ha la stessa topologia già discussa a proposito dei cicli, vale a dire (repetita iuvent [non è un errore, è CONGIUNTIVO]) **O** un INTERO AMBITO delimitato da graffe, e per ciò stesso di qualunque numero di linee di codice, **O** un'UNICA ESPRESSIONE chiusa col segno di punto e virgola. Desso segmento di codice sarà eseguito/valutato OVVERO DEL TUTTO IGNORATO secondo che, rispettivamente, *espressione*, ossia quello che si trova entro le parentesi, risulti booleanamente VERA o FALSA.  
*espressione* è quindi TOTALMENTE OMOLOGA, sia sul piano sintattico sia su quello funzionale, di quella che sta nelle parentesi di *while*, l'unica differenza essendo il numero di valutazioni che ne vengono fatte, ossia, nel caso presente **UNA SOLA**, a meno che, ma si tratta di un'ovvietà, *if* stessa non si trovi in un ciclo, nel qual caso *espressione* sarebbe bensì valutata numerose volte, ma sempre UNA VOLTA SOLA per ogni passaggio nel ciclo; NON MI VENITE A PARLARE DI cicli *if*, come purtroppo mi capita di sentire spessissimo.  
  
Il linguaggio prevede, come scelta OPZIONALE, la possibilità di inserire un segmento di codice alternativo a quello sottoposto a condizione tramite *if*, da eseguire appunto quando *espressione* risulti FALSA. Se tale segmento alternativo viene fornito, DEVE essere posto IMMEDIATAMENTE DOPO il suo *if* corrispettivo, e il suo segmento di codice annesso, facendolo precedere dalla parola di vocabolario *else*. Pertanto la sintassi è:

if( espressione ) ........... // segmento di programma  
else ........... // segmento alternativo

segmento alternativo ha anch'esso la consueta topologia e va sottolineato ancora una volta che la sua presenza, assieme a quella di *else* che lo introduce, è SOLO OPZIONALE, mentre dovrebbe essere del tutto evidente che se in un programma C++ compare un *else* NON PRECEDUTO da un *if* corrispondente, il compilatore segnalerà ERRORE senza pietà.  
In altre parole un *if* vive benissimo anche senza un *else*: è vocazionalmente un *single*; mentre un *else* senza il suo *if* non può stare: è vocazionalmente portato alla convivenza.  
L'ultima affermazione è talmente vera da poter ulteriormente essere rafforzata dicendo che *else* è addirittura monogamo e territoriale nel senso che si accoppia con UN SOLO *if* e precisamente con quello che gli si trova più vicino, come si evincerà leggendo i seguenti esempi.

**Esempio 1**  
  
int k;   
cin >> k;   
cout <<"mi hai fatto leggere un numero";  
if( k >= 0 ) cout <<" non";  
cout <<" negativo\n";

Questo esempio è il trionfo della litote: se si digita 1 sulla tastiera l'espressione in *if* risulta VERA e quindi viene eseguita *cout <<" non";* producendo come effetto la risposta congrua da parte del programma *"mi hai fatto leggere un numero non negativo"*; invece, se si digita -1, la FALSITÀ dell'espressione in *if* farà sì che sia IGNORATA la scrittura della stringa *" non"* e quindi, ancora una volta in modo coerente, il programma scriverà *"mi hai fatto leggere un numero negativo"*. Si noti che con questa formulazione non è stato necessario alcun *else*.

**Esempio 2**  
  
int k;   
cin >> k;   
cout <<"mi hai fatto leggere un numero";  
if( k >= 0 ) cout <<" positivo";  
else cout <<" negativo";  
cout <<'\n';

Questo esempio è equivalente al precedente, ma stavolta la formulazione richiede la presenza di un *else* per far in modo che anche il penultimo *cout* sia eseguito *sub condicione* piuttosto che incondizionatamente come prima. Se si eliminasse dall'esempio *else* si otterrebbe, in risposta alla digitazione di 1, il messaggio contraddittorio *"mi hai fatto leggere un numero positivo negativo"*, mentre grazie a *else*, e sempre in presenza di 1, il penultimo *cout* è IGNORATO; coerentemente sarebbe IGNORATO *cout <<" positivo";* se si digitasse -1, e in quel caso si eseguirebbe *cout <<" negativo";*.  
  
Ora provate a eseguire TRE volte il seguente **Esempio 3** digitando a ogni esecuzione rispettivamente 1, 0 e -1

**Esempio 3**  
  
int k;   
cin >> k;   
cout <<"mi hai fatto leggere un numero";  
if( k >= 0 ) if( k > 0 ) cout <<" positivo";  
else cout <<" negativo";  
cout <<'\n';

Che cosa avete notato? Se l'intento del programmatore fosse stato, come è lecito ipotizzare, quello di non farsi scrivere né *" positivo"* né *" negativo"* in caso di digitazione dello 0 (zero), la soluzione adottata è completamente peregrina, anche se a prima vista può sembrare giusta, specialmente a un neofita quali voi siete, perché non tiene conto della monogamia e territorialità di *else* che, nell'ultimo codice, si SPOSA con *if(k>0)* che gli sta più vicino del precedente *if(k>=0)*. Quest'ultimo quindi, per sua fortuna o disgrazia non è dato sapere, RESTA *single*. Ciò dovrebbe spiegarvi gli esiti che avete ottenuto, sempre ammesso che abbiate obbedito al suggerimento di eseguire il codice.  
È questa la ragione per cui molti, che non si fidano poi così tanto della propria logica, preferiscono aprire ambiti anche quando non servirebbe; in effetti, fermo restando che la logica è sbagliata, il codice precedente equivarrebbe a questo:

**Esempio 3 equivalentemente errato**  
  
int k;   
cin >> k;   
cout <<"mi hai fatto leggere un numero";  
if( k >= 0 )   
{  
if( k > 0 ) cout <<" positivo";  
else cout <<" negativo";  
}  
cout <<'\n';

Per il compilatore, RIPETO, è ESATTAMENTE LA STESSA COSA, ma forse l'occhio umano di un programmatore dotato di logica difettosa è aiutato a vedere dove sta il difetto. Né si pensi che il rimedio consista nella mera sostituzione dell'operatore *>=* col semplice operatore *>* nell'**Esempio 3** originale, perché neppure in tal caso si conseguirebbe il risultato voluto e sperato: provare per credere, ma direi che sia abbastanza ovvio. La soluzione VERA è pertanto questa:

**Esempio 3 GIUSTO**  
  
int k;   
cin >> k;   
cout <<"mi hai fatto leggere un numero";  
if( k > 0 ) cout <<" positivo";  
else if( k < 0 ) cout <<" negativo";  
cout <<'\n';

ossia l'inserimento di un ulteriore *if* appropriato DENTRO IL SEGMENTO DI CODICE INTRODOTTO DA *else*. In questo modo la digitazione di 0 provoca l'esclusione della scrittura di entrambi gli aggettivi, che sono scritti ciascuno SOLO quando occorra. Se poi non dovesse piacere nemmeno vedersi rispondere solo *"mi hai fatto leggere un numero"* alla digitazione di 0, si potrebbe anche apportare quest'ulteriore aggiunta:

**Esempio 3 e basta!**  
  
int k;   
cin >> k;   
cout <<"mi hai fatto leggere un numero";  
if( k > 0 ) cout <<" positivo";  
else if( k < 0 ) cout <<" negativo";  
else cout <<" nullo";  
cout <<'\n';

Siccome nulla vieta a OGNI segmento introdotto da un *else* di contenere un ulteriore *if* si capisce abbastanza facilmente che l'anafora presente nell'ultima versione del codice può essere protratta quanto si vuole. Considerate la seguente variante, in cui compare anche l'operatore unario *!* (punto esclamativo) col significato di negazione booleana (**IMPARATELO!**):

**Esempio 4**  
  
int k;   
cin >> k;   
cout <<"Per i miei gusti, ho letto un numero";  
if( k > 100 ) cout <<" positivo e GRANDE";  
else if(k > 10 ) cout <<" positivo e MEDIO";  
else if(k > 0 ) cout <<" positivo e PICCOLO";  
else if( ! k ) cout <<" NULLO";  
else if(k > -10 ) cout <<" negativo e PICCOLO";  
else if(k > -100 ) cout <<" negativo e MEDIO";  
else cout <<" esageratamente negativo";  
cout <<'\n';

Dato che, ed è già stato detto, OGNI *else* si sposa con l'*if* a lui più vicino, qui si realizza una catena ininterrotta di SEI *if*, ciascuno coniugato con l'*else* che lo segue IMMEDIATAMENTE; in questo caso l'uso degli ambiti inutili è un rimedio assai peggiore del male perché comporterebbe dovere scrivere il codice equivalente così

if( k > 100 ) cout <<" positivo e GRANDE";  
else {  
  if(k > 10 ) cout <<" positivo e MEDIO";  
  else {  
    if(k > 0 ) cout <<" positivo e PICCOLO";  
    else {  
      if( ! k ) cout <<" NULLO";  
      else {  
        if(k > -10 ) cout <<" negativo e PICCOLO";  
        else {  
          if(k > -100 ) cout <<" negativo e MEDIO";  
          else cout <<" esageratamente negativo";  
          }  
        }  
      }  
    }  
  }

che finisce con essere più complicato anche per l'occhio umano. In qualsiasi modo il codice sia scritto si dovrebbe capire che sarà eseguito UNO e UNO SOLO dei segmenti di codice sottoposti a condizione, precisamente il PRIMO che comporterà una valutazione come VERA dell'espressione tra parentesi. Le parole il PRIMO non sono del tutto oziose, poiché un valore digitato pari a 200 le rende TUTTE VERE, ma SOLO il segmento di codice che scrive sul canale di *output* la frase *" positivo e GRANDE"* sarà eseguito e NESSUN ALTRO, come è giusto che sia.  
Si osservi che È TUTTA RESPONSABILITÀ del programmatore ordinare le condizioni secondo logica; al compilatore non può interessare nulla se la logica di chi programma è **talmente difettosa da scrivere il precedente codice *a rovescio***, ossia mettendo per prima la condizione *k>-100* e per ultima quella *k>100*: l'accetterà ugualmente senza battere ciglio, al massimo pensando fra sé e sé: *adesso saranno affari tuoi*...  
Un'altra cosa da aggiungere è che la frase sarà eseguito UNO e UNO SOLO (citazione da poche righe sopra) è VALIDA, nel presente contesto, perché c'è un ULTIMO *else* INCONDIZIONATO. Se questo mancasse potrebbe anche accadere che NON SIA ESEGUITO UN BEL NULLA, ad esempio digitando appunto -100.  
  
Eventuali dichiarazioni inserite nei segmenti di programma sottoposti a condizione, sia introdotti da *if* sia da *else*, sono LIMITATE A QUEL SEGMENTO **ANCHE SE NON VIENE ESPLICITAMENTE APERTO UN AMBITO**. Per intendersi una presunta furbata come la seguente

int i;  
cin >> i;   
if(i) double k;  
else int k;  
// qui k è NON DICHIARATO

in cui si pretenderebbe di dichiarare diversamente una variabile secondo il valore fornito da tastiera per un dato, NON FUNZIONA, come spiega esaurientemente il commento, pur venendo compilata senza errori.  
  
La seconda istruzione condizionale prevista dal linguaggio è introdotta dalla parola di vocabolario *switch* con la seguente sintassi:

switch( espressione ) ...... // segmento di programma

che è solo apparentemente identica a quella di *if* (cfr. sopra). Le differenze sostanziali sono:

* il segmento di programma che segue l'istruzione ha una topologia completamente diversa ed esclusiva di questa istruzione;
* l'*espressione* che sta dentro le parentesi può essere **SOLO** di tipo **INTERO** o di tipo **ENUMERAZIONE**, NON di tipo booleano , come era sia per *if* sia per *while*, OPPURE può essere la dichiarazione immediatamente inizializzata di una singola variabile appartenente ai tipi citati, e che resterà confinata, come d'abitudine, al segmento di programma.

Il primo punto del precedente elenco NON significa che il compilatore non accetti segmenti di programma che abbiano la consueta topologia; significa piuttosto che tali segmenti sono quasi del tutto INUTILI. Per intendersi un costrutto come il seguente

**Esempio 0 (INUTILE)**  
  
int k;  
cin >> k;  
switch(k) cout << k << '\n';

viene tranquillamente compilato, ma l'istruzione *cout << k << '\n';* NON SARÀ MAI ESEGUITA, ma proprio **MAI**, e l'unico effetto collaterale di un simile codice potrebbe provenire, AL MASSIMO, dalla valutazione dell'espressione entro le parentesi tonde; nel caso presente, NESSUNO.  
  
Quanto al secondo punto dell'elenco, l'esclusione del tipo *bool* non significa che sia VIETATO utilizzarlo, ma solo che, allora, non si vede perché non usare semplicemente *if*. Peraltro, quando sarete cresciuti, vedrete che potrete metterci anche tipi molto sofisticati, ossia creati da voi, purché li provvediate di una regola chiara di conversione ai tipi interi.  
  
E giusto a proposito di tipi CREATI DA VOI, mette conto spendere qualche parola sulle *enumerazioni*, citate nello stesso punto dell'elenco, che saranno il PRIMO di tali tipi che incontrerete sul vostro cammino di conoscenza; qui ci si limiterà a quanto sia essenziale per l'uso con *switch*, procrastinando i dettagli fino alla stesura dell'appropriato canto.  
  
La LOGICA dice che, se un'enumerazione è un tipo, deve servire a poter dichiarare variabili di QUEL tipo; d'altronde, se è un tipo CREATO dal programmatore, ci DEVE ESSERE un modo convenuto per crearlo, la creazione DEVE AVVENIRE PRIMA di poter effettuare qualsiasi dichiarazione che lo coinvolga e deve avvenire NELLO STESSO AMBITO ovvero IN UN AMBITO PIÙ ESTERNO rispetto a quello in cui si vuole effettuare la dichiarazione.  
  
La creazione del tipo, PER ORA, avverrà con l'ausilio della parola di vocabolario *enum*, nell'ambito globale, così da poter essere fruita all'interno di qualsiasi funzione scritta successivamente, con la seguente sintassi:

enum   nome\_del\_tipo   
{nome1, nome2, nome\_etcetera};

Se non fosse per la mancanza di parentesi quadre, assomiglierebbe alla dichiarazione di un *array* inizializzato. Qui *nome\_del\_tipo* è appunto il nome del tipo creato, da usare quindi in ogni successiva dichiarazione che lo voglia sfruttare; all'interno delle graffe, obbligatorie, trova posto una lista di nomi separati da virgole e in numero arbitrario che sono, a tutti gli effetti, delle **costanti dichiarate** (andare a rivedere che cosa significa) appartenenti al cosiddetto tipo sottostante la *enum* e **visibili** in OGNI ambito interno a quello in cui la *enum* è stata creata; quindi OVUNQUE, se, come detto, si è sfruttato l'ambito globale.  
Il tipo sottostante la *enum* è il tipo *int*, opportunamente e tacitamente modificato dal compilatore stesso per potervi accomodare eventuali valori espliciti con cui il programmatore voglia inizializzare qualcuna delle costanti; ad esempio in

enum   nome\_del\_tipo   
{nome1, nome2, nome\_etcetera = 0xffffffffffffffffULL};

il tipo sottostante è *unsigned long long*.  
Ogni variabile dichiarata appartenente al tipo *nome\_del\_tipo* può essere inizializzata **SOLO** e **SOLTANTO** con il valore di una delle costanti dichiarate nella stessa *enum*, cui si acceda usando l'appropriato nome, come già detto, VISIBILE.  
  
Tornando a *switch*, la topologia del segmento di programma idonea all'uso prevede che vi appaiano delle cosiddette *etichette*; ALMENO UNA: ogni *etichetta* è costituita **O** dalla parola di vocabolario *default*, terminata col carattere ':' (due punti) **O** dalla parola di vocabolario *case*, seguita da una costante dello stesso tipo dell'espressione o della variabile situata nelle parentesi tonde, o ad esso convertibile, e ancora conclusa col carattere ':' (due punti). Ecco alcuni esempi:

**Esempio 1 (INUTILE)**  
  
int main( ){  
enum settimana   
{domenica, lunedi, martedi, mercoledi,  
giovedi, venerdi, sabato};  
settimana giorno = domenica;  
switch(giorno)  
default: cout <<"che giorno è ?\n";}

Quest'esempio è il contrario dell'**Esempio 0**, nel senso che, stavolta, l'espressione *cout <<"che giorno è ?\n";* è SEMPRE ESEGUITA, ma proprio **SEMPRE**, e quindi è altrettanto inutile aver allestito tutto l'*ambaradan* precedente. L'unico pregio dell'esempio è aver introdotto una *enum* alquanto sensata, che dovrebbe farne capire l'utilità.

**Esempio 2 (un po' UTILE)**  
  
int main( ){  
enum settimana   
{domenica, lunedi, martedi, mercoledi,  
giovedi, venerdi, sabato};  
settimana giorno = domenica;  
switch(giorno)  
case domenica: cout <<"oggi è domenica\n";}

Qui l'espressione *cout <<"oggi è domenica\n";* continua a essere eseguita, come prima, ma vedrete che se sostituirete l'inizializzazione di *giorno* con un'altra costante, **NULLA PIÙ** sarà scritto.

**Esempio 3 (ancora più UTILE)**  
  
enum settimana   
{domenica, lunedi, martedi, mercoledi,  
giovedi, venerdi, sabato};  
settimana funz(  )  
{settimana giorno;  
// giorno = ....  
// chiedere al calcolatore la data corrente   
// e inizializzare coerentemente *giorno*  
return giorno;}  
  
int main(  )  
{switch(settimana giorno = funz(  ))  
case domenica: cout <<"oggi è domenica\n";}

Questo esempio funziona, SENZA MODIFICHE, in qualunque giorno della settimana, annunciando il giorno festivo SOLO se è effettivamente domenica: si suppone, ovviamente, che l'orologio interno del calcolatore sia ben regolato. A farlo comportare correttamente provvedrà la funzione *funz(  )* col cui risultato *main* inizializza la propria variabile *giorno* adeguatamente dichiarata direttamente entro le parentesi di *switch*; tutto il resto è come già spiegato. Come ricavare la data corrente dal calcolatore non è materia da affrontare per adesso: sappiate solo che si può (GRAN LINGUAGGIO, ma lo fanno anche altri); se volete solo essere certi che funzioni provate semplicemenete a sostituire *return giorno;* con *return domenica;* o con *return giovedi;*.

**Esempio 4 (una delle forme più comuni)**  
  
enum settimana   
{domenica, lunedi, martedi, mercoledi,  
giovedi, venerdi, sabato};  
settimana funz(  )  
{settimana giorno;  
// giorno = ....  
// chiedere al calcolatore la data corrente   
// e inizializzare coerentemente *giorno*  
return giorno;}  
  
int main(  )  
{cout <<"oggi è ";  
switch(settimana giorno = funz(  ))  
{case domenica:  
cout <<"domenica, che viene prima di ";  
case lunedi:   
cout <<"lunedi, che viene prima di ";  
case martedi:   
cout <<"martedi, che viene prima di ";  
case mercoledi:  
cout <<"mercoledi, che viene prima di ";  
case giovedi:  
cout <<"giovedi, che viene prima di ";  
case venerdi:  
cout <<"venerdi, che viene prima di ";  
case sabato:  
cout <<"sabato, che viene prima di domenica ";  
default: cout <<"e la settimana di otto giorni non esiste.";  
} cout << '\n';}

Osservate che, come sempre è successo, la volontà/necessità del programmatore di inserire più di UN SOLO punto e virgola nel segmento di programma sotto condizione da parte di *switch* HA PRETESO l'apertura di un ambito. ESEGUITE SUI DUE PIE' QUESTO CODICE, facendo restituire a *funz( )*, in alternanza, i diversi giorni della settimana e CAPITE QUEL CHE ACCADE.  
  
Dall'esecuzione di quest'ultimo codice (L'AVETE ESEGUITO?) si apprende quanto segue:

* quando il programma, in corso di esecuzione, incontra uno *switch* l'esecuzione stessa prosegue a partire dal punto di codice marcato con l'etichetta *case* contraddistinta dal valore della costante che coincide col valore dell'espressione valutata entro le parentesi dello *switch* stesso;
* Se NESSUNA delle costanti che concorrono alle etichette *case* coincide col valore dell'espressione valutata in *switch*, allora l'esecuzione riprende dal punto di codice marcato con l'etichetta *default*;
* Nelle identiche condizioni del punto precedente, se l'etichetta *default* non c'è (OGNI ETICHETTA È SOLO OPZIONALE), allora non viene eseguito UN BEL NULLA e l'esecuzione SALTA A PIE' PARI l'intero costrutto *switch*;
* i due punti precedenti spiegano esaurientemente entrambi gli esempi denominati INUTILI;
* le etichette, in quanto tali, sono assolutamente trasparenti all'esecuzione: tutte quelle eventualmente successive a quella che marca il punto di ripresa dell'esecuzione stessa vengono attraversate come aria fresca;
* **È UN ERRORE**, sempre fatale alla riuscita della compilazione, inserire DUE ETICHETTE *case* recanti la STESSA COSTANTE;
* è del tutto intuibile, per quanto detto fin qui, che l'ORDINE IN CUI SONO DISPOSTE le etichette nel segmento di programma sottoposto a controllo da parte si *switch* HA GRANDE RILEVANZA per la logica del programma, come si capisce facilmente anche dall'esempio fornito: nessuno ha mai visto il martedì precedere il lunedì, NELLA STESSA SETTIMANA. Del resto si rammenta a tutti che l'**ORDINE** è frutto della musicalità, cioè di una delle virtù del buon programmatore;
* l'unica deroga a quanto affermato al punto precedente si ha quando il programmatore ricorre all'uso della parola di vocabolario *break*, inserendola in punti strategici del segmento di programma controllato da *switch*: quando l'esecuzione incontra tale parola il segmento di programma sotto controllo di *switch* viene istantaneamente interrotto (è ciò che *break* significa in angloamericano) e l'esecuzione SALTA al punto di codice immediatamente successivo all'intero costrutto *switch*.

Come esempio di quanto detto nell'ultimo punto del precedente elenco eccovi il codice seguente:

int k;  
cin >> k;  
switch(k)  
{  
case 0: cout <<"sto eseguendo case 0\n";  
break;  
case 1: cout <<"sto eseguendo case 1\n";  
break;  
default: cout <<"non ho trovato alcun case\n";  
}   
cout <<"qui sono fuori dallo switch\n";

I messaggi scritti dal programma sono autoesplicativi per lo svolgersi dell'esecuzione: si noti che in un caso del genere la disposizione relativa delle etichette è DEL TUTTO IRRILEVANTE e che il costrutto *switch* finisce per assomigliare al seguente, nel senso che ha lo stesso EFFETTO, con, al massimo, la valutazione di un'espressione in più [*!(k-1)*], senza peraltro **alcun effetto collaterale**; ESEGUITE E VEDRETE:

int k;  
cin >> k;  
if(!k) cout <<"sto eseguendo case 0\n";  
else if(!(k-1)) cout <<"sto eseguendo case 1\n";  
else cout <<"non ho trovato alcun case\n";  
cout <<"qui sono fuori dallo switch\n";

NON BISOGNA PERÒ GENERALIZZARE: **NON SEMPRE** uno *switch* in cui compare *break* equivale a una catena *if/else* come quella mostrata sopra, e per le seguenti ragioni LOGICHE:

* innanzitutto, e questa è un'ovvietà, l'equivalenza può sussistere SOLO se **TUTTE LE ETICHETTE** dello *switch* sono terminate con *break*; appena ce ne fossero due consecutive SENZA *break* frammezzo, l'equivalenza non può, LOGICAMENTE, essere più proposta;
* ma anche nel caso in cui NEPPURE un *break* manchi all'appello uno *switch* e una catena *if/else* potrebbero NON ESSERE AFFATTO equivalenti, come mostra il seguente controesempio:

int funz(int k)  
{return k+1;}  
  
int main(  )  
{  
int n;  
cin >> n;  
switch(funz(n))  
{  
case 0: cout <<"case 0\n"; break;  
case 1: cout <<"case 1\n"; break;  
case 2: cout <<"case 2\n";  
}}

Non c'è modo di sostituire questo *switch* con una catena *if/else* equivalente, per la semplice ragione che *if* valuta l'equivalente booleano del valore restituito dalla funzione, e quando *funz(int k)* restituisca qualsiasi valore diverso da zero per *if* è sempre la stessa cosa. Né giova escogitare il seguente marchingegno

int funz(int k)  
{return k+1;}  
  
int main(  )  
{  
int n, c;  
cin >> n;  
if((c = funz(n)) == 0) cout <<"case 0\n";  
else if(c == 1) cout <<"case 1\n";  
else if(c == 2) cout <<"case 2\n";  
}}

che ha bensì lo stesso effetto, A PRIMA VISTA, ma ha anche, stavolta, **effetti collaterali**: la dichiarazione e inizializzazione della variabile *c*, che a *switch* non abbisognano. IMPARATE l'operatore di confronto *==* e **NON CONFONDETELO MAI** con l'operatore di assegnamento *=* **[NON AVETE IDEA DI QUANTE VOLTE LI SBAGLIERETE]**.  
  
Un'ulteriore cosa da dire è che **È VIETATO** effettuare dichiarazioni DOPO un'etichetta, A MENO CHE NON LE SI CONFINI IN UN AMBITO APPOSITAMENTE CREATO. Per intendersi si dànno qui appresso due esempi, il primo SBAGLIATO e il secondo CORRETTO:

**Esempio sbagliato**  
  
int n;  
cin >> n;  
switch(n)  
{int q; // qui è corretto  
case 1:  
int z = 1; // ERRORE  
q = 2, cout << "case 1 " << q << ' ' << z << '\n';  
break;  
case 0: q = 4, cout << "case 0 " << q << '\n';  
}   
  
**Esempio corretto**  
  
int n;  
cin >> n;  
switch(n)  
{int q; // qui è corretto  
case 1:  
{ // APERTURA VOLONTARIA DI AMBITO  
int z = 1; // CORRETTO  
q = 2, cout << "case 1 " << q << ' ' << z << '\n';  
} // CHIUSURA DELL'AMBITO  
// VOLONTARIAMENTE APERTO  
break;  
case 0: q = 4, cout << "case 0 " << q << '\n';  
}

La parola *break* non serve solo a uscire prima del tempo dal segmento di programma governato da uno *switch*, ma anche a uscire prematuramente da un ciclo: sono anzi queste le uniche collocazioni ammissibili per tale parola di vocabolario, nel senso che il compilatore segnalerà ERRORE ogni volta che *break* sia incontrato al di fuori di questi contesti.  
Quando *break* ricorre in un ciclo, procura al programmatore una valida via d'uscita da OGNI CICLO, ma in particolare da un ciclo che fosse stato impostato come ciclo ETERNO; anzi, in tal caso, è l'UNICO MODO di uscirne senza che si debba incorrere in qualche errore di esecuzione.   
Si consideri il seguente esempio ERRATO:

int i;  
while(true)  
{  
cout << "digita un intero ", cin >> i,  
cout << "eccotene il quadrato " << i\*i << '\n';  
}

L'esempio non è errato perché fa cose sbagliate, ma solo perché non smette mai di fare cose giuste: non c'è modo di dissuadere il programma dal continuare imperterrito a domandare numeri da elevare al quadrato, dato che l'espressione in *while* è, per costruzione, LETTERALMENTE e COSTANTEMENTE **VERA**. Si tratta appunto di un ciclo ETERNO.  
È già stato detto che non bisogna temere i cicli eterni che, anzi, in molte situazioni sono una mano santa; basta solo prevedere una sensata via d'uscita, come qui appresso:

int i;  
while(true)  
{  
cout << "digita un intero ", cin >> i;  
if(!i) break;  
cout << "eccotene il quadrato " << i\*i << '\n';  
}  
cout << "Addio!\n";

Con questo codice si possono passare, se si vuole, anche due ore di seguito a calcolare quadrati, ma quando non se ne potrà più sarà sufficiente digitare 0, del cui quadrato, sinceramente, si riesce facilmente a fare a meno, per piantarla e ricevere dal programma un commiato forse un po' fatalista, ma comunque educato.  
  
Oltre a *break*, che, come si è visto, può abitare tanto in cicli quanto in *switch*, esiste un'altra parola di vocabolario, ***continue***, che può trovarsi SOLO nei cicli, ove svolge un ruolo importantissimo; si consideri la seguente variante del precedente codice, in cui si è SOLO richiesto, per pura comodità, di avere numeri abbastanza piccoli da maneggiare:

int i;  
while(true)  
{  
cout << "digita un intero tra 0 e 15 ", cin >> i;  
if(!i) break;  
else if(i > 15 || i < 0) continue;  
cout << "eccotene il quadrato " << i\*i << '\n';  
}  
cout << "Addio!\n";

**ESEGUITE QUESTO CODICE E PROVATE A NON OBBEDIRGLI**  
  
L'effetto di *continue*, nel caso presente, consiste nel rifiuto di numeri fuori intervallo (IMPARATE l'operatore *||* che esegue l'OR LOGICO) e questo si ottiene poiché, quando si esegue *continue*, viene SALTATO tutto quanto si trova tra *continue* stesso e la fine del ciclo, che riprende daccapo con l'iterazione successiva, lungi dall'essere definitivamente interrotto come accade con *break*. Del resto *continue* non significa di certo *interrompi*.  
Si ricorda a questo punto che, se il ciclo è gestito dalla parola *for* nella sua prima realizzazione, l'eventuale esecuzione di *continue* al suo interno NON IMPEDISCE che, PRIMA DI PASSARE ALL'ITERAZIONE SUCCESSIVA, sia comunque valutata/eseguita la terza espressione che sta nelle parentesi di *for*, qualora presente.  
Si ricorda altresì che il ciclo gestito dalla parola *for* nella sua seconda realizzazione è l'UNICO ciclo che, per propria natura, NON si presta a realizzare un ciclo eterno.

**Canto dodicesimo: operatori, precedenze, associatività, sequenzialità e *pietre miliari*; parte I: gli operatori**

Questo argomento è piuttosto vasto, come si intuisce dal titolo, ed è di GRANDE importanza per la piena comprensione dei meccanismi fini del funzionamento di un programma, la qual cosa evidentemente giova a scriverne di efficienti e produttivi.  
Proprio per la sua ampiezza e importanza sarà suddiviso in diversi canti, dei quali questo è il primo; si comincia col semplice ELENCO di TUTTI gli operatori del linguaggio, parecchi dei quali sono già stati incontrati strada facendo, essendo impossibile scrivere esempi di codice sensati che ne fossero del tutto privi; gli operatori sono divisi in categorie, per affinità tra di loro, e per ciascuno è dato un esempio del suo uso in un'espressione in cui compaia DA SOLO, e quindi, MOLTO SPESSO, inutile.

|  |  |  |
| --- | --- | --- |
| Operatori aritmetici unari (UN SOLO OPERANDO) | | |
| incremento (prefisso) | ++ | int a = 1; ++a; |
| decremento (prefisso) | -- | int a = 1; --a; |
| incremento (suffisso) | ++ | int a = 1; a++; |
| decremento (suffisso) | -- | int a = 1; a--; |
| segno positivo | + | int a = 1; +a; |
| segno negativo | - | int a = 1; -a; |
| negazione binaria (**SOLO OPERANDO INTERO**) | ~ | int a = 1; ~a; |

|  |  |  |
| --- | --- | --- |
| Operatori aritmetici binari (DUE OPERANDI) | | |
| addizione | + | int a=3, b=3; a+b; |
| sottrazione | - | int a=3, b=3; a-b; |
| moltiplicazione | \* | int a=3, b=3; a\*b; |
| divisione | / | int a=3, b=3; a/b; |
| resto (**SOLO OPERANDI INTERI**) | % | int a=3, b=3; a%b; |
| and binario (**SOLO OPERANDI INTERI**) | & | int a=3, b=3; a&b; |
| or binario (**SOLO OPERANDI INTERI**) | | | int a=3, b=3; a|b; |
| xor binario (**SOLO OPERANDI INTERI**) | ^ | int a=3, b=3; a^b; |
| scorrimento a destra (**SOLO OPERANDI INTERI**) | >> | int a=3, b=3; a>>b; |
| scorrimento a sinistra (**SOLO OPERANDI INTERI**) | << | int a=3, b=3; a<<b; |

**Osservazioni in corso d'opera:**

* Salvo ove esplicitamente annotato TUTTI i precedenti operatori possono avere operandi di qualsiasi tipo nativo (significa NON CREATO dal programmatore), eccetto OVVIAMENTE *void*. Se si tratta di operatori unari (incrementi, decrementi o segni) non si pone, evidentemente, alcun problema di compatibilità; per gli operatori binari il compilatore promuove automaticamente e tacitamente un eventuale operando più debole al tipo di quello più forte, che sarà anche il tipo del risultato, mentre se gli operandi sono dello STESSO tipo l'operazione avviene NEL RISPETTO DEL TIPO. In definitiva, ad esempio, la divisione di un *int* che vale 1 per UN ALTRO *int* che valga 2 ha per risultato un *int* che vale 0, mentre la divisione di un *int* che vale 1 per un *float* che valga 2 ha come risultato un float che vale 0.5. Nessuna di queste due situazioni è migliore dell'altra, perché **ENTRAMBE** possono essere utilmente sfruttate dal buon programmatore.
* Gli operatori *>>* e *<<*, che finora avevamo sempre incontrato rispettivamente come operatori di lettura e scrittura di dati, appaiono qui sotto una diversa veste e sono denominati operatori di scorrimento, quando i loro operandi sono di tipo riconducibile al tipo *int*, al lordo di ogni possibile modificatore.  
  Si tratta del primo caso in cui riscontrate la TOTALE DIVERSITÀ DI COMPORTAMENTO di un operatore secondo il tipo dei suoi operandi, ANZI, PIÙ PRECISAMENTE, quando si tratti di operatori binari, secondo il tipo **DELL'OPERANDO SINISTRO** (infatti *cin* e *cout* NON SONO CERTO DEGLI *int*).  
  Accendete (e TENETE ACCESO) un segnalatore nel vostro cervello a proposito di questo fatto, perché, andando avanti, lo ritroverete e ne comprenderete definitivamente **L'ENORME PORTATA** (GRAN LINGUAGGIO).
* Ciò che fanno gli operatori di scorrimento e tutti gli operatori che recano nella propria descrizione l'aggettivo binario (che, nel contesto in cui viene usato, NON C'ENTRA NULLA col numero di operandi) è spiegato dettagliatamente nelle pagine dedicate a ciascun operatore: non è il caso che andiate a leggerle ora. Quello che fa l'operatore *%* è implicato nel nome stesso usato per descriverlo.
* Di TUTTE le espressioni campione riportate come esempi d'uso dei diversi operatori, **SOLTANTO QUELLE** che concernono gli incrementi e i decrementi hanno effetti sull'operando coinvolto e, pertanto, possono a buon diritto apparire, così come sono, in un codice vero; **TUTTE LE ALTRE** lasciano **INTONSI** il/i proprio/ii operando/i, e pertanto, non avendo alcun altro effetto collaterale, se ricopiate così come sono in un codice vero, sono ritenute bensì corrette dal compilatore, ma sono prive di senso per il programma.
* Dovrebbe essere superfluo sottolineare che gli operatori che hanno lo stesso segno grafico si riconoscono come unari o binari per il contesto in cui sono usati... e questo vale anche per quelli che si incontreranno appresso.

|  |  |  |
| --- | --- | --- |
| Operatori di assegnamento (BINARI PER NATURA) | | |
| assegnamento | = | int a=3, b; b=a; |
| addizione con assegnamento | += | int a=3, b=3; b+=a; |
| sottrazione con assegnamento | -= | int a=3, b=3; b-=a; |
| moltiplicazione con assegnamento | \*= | int a=3, b=3; b\*=a; |
| divisione con assegnamento | /= | int a=3, b=3; b/=a; |
| resto con assegnamento | %= | int a=3, b=3; b%=a; |
| and binario con assegnamento | &= | int a=3, b=3; b&=a; |
| or binario con assegnamento | |= | int a=3, b=3; b|=a; |
| xor binario con assegnamento | ^= | int a=3, b=3; b^=a; |
| scorrimento a destra con assegnamento | >>= | int a=3, b=3; b>>=a; |
| scorrimento a sinistra con assegnamento | <<= | int a=3, b=3; b<<=a; |

**Osservazioni in corso d'opera:**

* al primo di questi operatori è già stato dedicato un intero canto, e quindi non si pensa di scrivere altro;
* per quanto riguarda ciascuno degli altri, usando *+=* come campione, quando si è detto che   
  *b+=a;*  
  equivarrebbe a  
  *b = b + a;*  
  senza fare altrettanto schifo, dovrebbe essere detto tutto: valgono, ovviamente, le **STESSE RESTRIZIONI DI TIPO** che si applicano agli operatori omologhi nella loro versione SENZA assegnamento;
* per quano detto al punto precedente, stavolta **TUTTE LE ESPRESSIONI** campione sono sensate, dato che TUTTE modificano il valore del proprio operando sinistro;
* spero che sia superfluo sottolineare che all'interno del segno grafico dell'operatore non deve essere inserito ALCUNO SPAZIO, e questo vale per tutti gli operatori il cui segno grafico consti di più di un solo carattere.

|  |  |  |
| --- | --- | --- |
| Operatore logico unario | | |
| negazione booleana | ! | bool a = true; !a; |

|  |  |  |
| --- | --- | --- |
| Operatori logici binari | | |
| and booleano | && | bool a=true, b=true; a&&b; |
| or booleano | || | bool a=false, b=false; a||b; |

**Osservazioni in corso d'opera:**

* non esiste un operatore per l'*xor booleano*;
* nessuna delle espressioni campione ha senso;
* ognuno di questi operatori non richiede necessariamente operandi che appartengano al tipo *bool*, come indicato negli esempi, mercé il fatto che OGNI espressione valutabile ha un equivalente booleano; così, ad esempio, anche un'espressione come *!4* (in cui proprio appaia la costante esplicita 4, di tipo *int*) ha perfettamente senso e ha il valore (booleano) *false*;
* reciprocamente, quantunque il tipo del valore dell'espressione in cui compaiono questi operatori sia *bool*, tale valore può tranquillamente essere usato in un'espressione numerica, dandole un contributo pari a 0 (se *false*) o pari a 1 (se *true*); così, ad esempio, nei seguenti assegnamenti a una variabile *x* dichiarata di tipo *int*  
  *x = !4; x += (5||0);*  
  *x* finisce per acquisire il valore 1 (se avete capito perché, allora avete capito; la coppia di parentesi è stata messa per chiarezza; in realtà non era indispensabile).

|  |  |  |
| --- | --- | --- |
| Operatori di confronto (BINARI PER NATURA) | | |
| uguaglianza | == | int a=3, b=3; a==b; |
| disuguaglianza | != | int a=3, b=-3; a!=b; |
| maggiorazione | > | int a=3, b=-3; a>b; |
| minorazione | < | int a=-3, b=3; a<b; |
| non minorazione | >= | int a=3, b=3; a>=b; |
| non maggiorazione | <= | int a=3, b=3; a<=b; |

**Osservazioni in corso d'opera:**

* i due operandi di ciascuno di questi operatori DEVONO appartenere a tipi tra loro compatibili, con lo stesso livello di compatibilità già richiesto dagli operatori aritmetici o dall'operatore di assegnamento. Del resto domandarsi se un limone sia diverso o uguale nel confronto con un rasoio, appare francamente poco sensato;
* il risultato dell'operazione è SEMPRE di tipo *bool*, quindi è materia prima idonea per gli operatori logici discussi sopra; altresì il risultato booleano può ancora contribuire a valutazioni squisitamente numeriche, come in  
  *int x = 5 > 0;*
* per lunga esperienza vissuta **NON SARÀ MAI ABBASTANZA RIPETUTO E SOTTOLINEATO** che i due operatori *==* (operatore di confronto per uguaglianza) e *=* (operatore di assegnamento) **NON SONO LO STESSO OPERATORE E NON VANNO INTERSCAMBIATI**; il brutto è che il compilatore **QUASI MAI È IN GRADO DI SEGNALARVI ERRORE** se li usate a sproposito, per cui **SONO AFFARI VOSTRI** se li confondete. Dei due, quello che assomiglia maggiormente all'*uguale* della matematica è *==*, **NON** *=*;
* ecco, così ve l'ho detto ancora una volta, sperando che finalmente smettiate di sbagliarli (a meno che non sbagliate APPOSTA perché ve lo dico troppo spesso...in tal caso IGNORATE il punto precedente).
* ...e così ve l'ho detto altre due volte: metteteci un *break*...

|  |  |  |
| --- | --- | --- |
| Operatori UNARI per accesso alla memoria | | |
| reperimento di indirizzo | & | int a=3; &a; |
| dereferenza di puntatore | \* | int a[]{1}; \*a; |

**Osservazioni in corso d'opera:**

* questi due operatori unari hanno lo stesso segno grafico di due operatori binari (vedi sopra) che, con loro, c'entrano meno di quanto c'entrino queste note col kamasutra;
* quando questi due stessi segni grafici vengono usati in una dichiarazione, piuttosto che in un'espressione, il loro significato cambia radicalmente: in tale frangente il primo serve a dichiarare un *riferimento*, il secondo a dichiarare un puntatore;
* introducendo l'operatore unario *\** ci si è giovati di un array invece che di un puntatore: la cosa è equivalente perché si dovrebbe sapere (NULLA VA DIMENTICATO) che gli array non sono altro che puntatori costanti;
* e comunque le due espressioni campione mostrate, come si era anticipato e più volte osservato, sono corrette, ma senza alcun significato utile;
* ciò che questi due operatori fanno, o è già stato detto o è dettagliatamente discusso nelle pagine loro dedicate. Non andateci, ora; basti sapere che *&* accetta operandi di qualsiasi tipo, purché dotati di indirizzo (ad esempio non può essere applicato a costanti esplicite), mentre *\** accetta come operandi SOLO dei puntatori (di qualunque tipo);
* si può affermare che, in QUASI tutti i contesti, questi due operatori sono uno l'inverso dell'altro nel senso che, con *p* opportunamente dichiarato e inizializzato ogni volta, tanto *&\*p* quanto *\*&p* non sono altro che *p* stesso.

|  |  |  |
| --- | --- | --- |
| Operatori BINARI per accesso alla memoria | | |
| accesso a elemento di array o puntatore | [ ] | int a[]{1}; a[0]; |
| risolutore di ambito | :: | std::cout; |
| accesso a membro tramite oggetto | . | *futurum* |
| accesso a membro tramite puntatore/indirizzo | -> | *futurum* |
| puntatore a membro tramite oggetto | .\* | *futurum* |
| puntatore a membro tramite puntatore/indirizzo | ->\* | *futurum* |

**Osservazioni in corso d'opera:**

* *hic sunt leones*: questi operatori, eccettuati i primi due che si erano già incontrati, sono da rimandare a giorni più maturi: anche per questa ragione si sono tralasciate, negli ultimi quattro, perfino le espressioni campione. La loro presenza serve solo alla completezza dell'elenco.

|  |  |  |
| --- | --- | --- |
| Miscellanea di operatori | | |
| invocazione di una funzione | (*lista di parametri*) | funz(par1, par2, .......); |
| casting | (*nome\_di\_un\_tipo*) | int i; (double)i; |
| operatore virgola | , | int a, b; a, b; |
| operatore condizionale | ?    : | bool a; int b, c; a ? b : c |

**Osservazioni in corso d'opera:**

* nell'invocazione dell'esecuzione di una funzione, cosa che si è vista fin dall'inizio del presente percorso, la coppia di parentesi che racchiude obbligatoriamente la lista dei parametri trasmessi alla fuzione stessa, è considerata, nella grammatica del linguaggio, **un operatore**. Ciò è estremamente rilevante, come si comprenderà meglio procedendo nell'avventura dell'apprendimento del linguaggio. Per quanto riguarda il numero degli operandi, si può affermare che è imprecisato *a priori*.
* il *casting* è un operatore unario che consente di modificare (SOLO PROVVISORIAMENTE, e SOLO nell'espressione in cui ricorre) il tipo del suo operando, trasformandolo in un tipo compatibile. Utile, ad esempio, per forzare la promozione di tipo in una divisione tra variabili dichiarate come *int* quando fosse necessario non perdere la parte frazionaria del risultato, come in   
  *int i = 1, j = 2; cout << (double)i/j << '\n';*  
  per vedersi scrivere 0.5.
* un altro operatore insospettabile, nel senso che forse non è immediato pensare che lo sia, è la virgola, che separa una sottoespressione da un'altra all'interno di un'unica espressione. La sua utilità sarà chiarita dall'esperienza; per adesso, ad esempio, consente spesso di non dover per forza aprire un ambito per realizzare dei cicli o delle istruzioni condizionali, ma c'è comunque di meglio.
* dire che il segno grafico dell'operatore condizionale è costituito dalla coppia di caratteri punto interrogativo e due punti significa che quei due caratteri vanno usati come nell'espressione campione (comunque inutile) presentata, ossia che vanno separati da una sottoespressione. La sintassi è quindi, come si vede nell'esempio,  
  *operando1  ?  operando2  :  operando3*  
  si può quindi dire che si tratta di un operatore ternario (l'unico tale nel linguaggio), i cui tre operandi sono espressioni. La prima DEVE essere un'espressione VALUTABILE, e, come tale, viene SEMPRE valutata: secondo che il suo valore equivalente booleano risulti rispettivamente VERO o FALSO, sarà valutata/eseguita l'espressione *operando2* e IGNORATA l'espressione *operando3*, o VICEVERSA. Le due ultime espressioni, a loro volta, devono essere di tipo compatibile tra loro, compresa l'eventualità che siano entrambe NON valutabili. È ammesso anche che una delle due (indipendentemente dal tipo dell'altra), ovvero ENTRAMBE, sia/siano espressione/i *throw*. Che cosa sia un'espressione *throw* lo imparerete a suo tempo.  
  L'uso dell'operatore ternario condizionale è spesso l'unico modo di scrivere una funzione sensata che sia qualificabile come *constexpr*: anche questo vi sarà chiaro col tempo.

Sono poi previste alcune altre componenti che, nell'analisi grammaticale del linguaggio, sono classificate come operatori per i quali è addirittura coniata una parola di vocabolario dedicata. Uno di questi, l'operatore *new*, è già stato incontrato (tornate indietro a salutarlo, se non ve lo ricordate); qui si darà l'elenco SOLO NOMINATIVO di tutti questi operatori, giusto per poter dire che, dopo questo canto, li avete incontrati TUTTI, se non altro dal punto di vista onomastico. La discussione dettagliata su ciascuno si trova nelle pagine loro riservate, ovvero sarà ripresa al momento opportuno di questo percorso di conoscenza (abituatevi a tenere accesi i LED CEREBRALI che servono a ricordare le pendenze in atto).

|  |  |  |  |
| --- | --- | --- | --- |
| Lista degli operatori per cui esiste una parola riservata | | | |
| new | delete | sizeof | sizeof... |
| const\_cast | reinterpret\_cast | static\_cast | dynamic\_cast |
| typeid | noexcept | alignof | throw |

**Osservazione:** i tre puntini a destra di *sizeof* nella quarta celletta della precedente tabella fanno parte integrante del nome di quell'operatore, e quindi si tratta di un altro operatore rispetto a quello che lo precede in tabella.  
  
Lo standard 2011 ha anche introdotto una sorta di pseudo-operatore che consente al programmatore di definirsi dei suffissi personalizzati da poter posporre a costanti esplicite per modificarne il valore secondo convenienza: per intendersi qualcosa che faccia capire al compilatore che quando si scrivesse una costante esplicita in questo modo  
  
*180\_gradi*   
  
(che sarebbe un ERRORE, in condizioni normali) si intende che quella costante abbia il valore di pi greco. La sintassi necessaria ad attribuire questo significato al suffisso personalizzato *\_gradi* è, al momento, prematura: se ne discute nella pagina dedicata alle virgolette, reperibile alla voce *operatori del C++*.   
  
Questo canto termina segnalandovi che il compilatore riconosce anche alcuni digrammi o trigrammi attribuendo loro il significato di singoli caratteri che fossero assenti su tastiere non ASCII. Altresì riconosce grafìe alternative a quelle normali per operatori i cui segni grafici fossero anch'essi indisponibili su particolari tastiere (il mondo è grande e tutti hanno diritto di usare il C++). **VOI NON ABUSATE DI CIÒ** perché l'uso a sproposito di tali alternative può avere severe controindicazioni sulla sorgente del programma. Proprio per questo, dopo avervi raccontato, per completezza, che esistono, non ve ne scrivo nemmeno la lista, lasciandovi volutamente nell'ignoranza di questo argomento.

**Canto tredicesimo: operatori, precedenze, associatività, sequenzialità e *pietre miliari*; parte II: precedenze e associatività**

Se si è letto e ben compreso il canto precedente, si dovrà riconoscere che un'espressione che contenga UN SOLO OPERATORE ha senso in pochissimi casi: segnatamente SOLO se si tratta **O** di uno degli operatori di incremento/decremento **OPPURE** dell'operatore di assegnamento, o da solo o in una delle sue forme composite (*+=* et similia).  
Ne segue, per pura deduzione logica, che la preponderante maggioranza delle espressioni sensate contiene più di un solo operatore, se non altro perché l'operatore di assegnamento (che conta pure come UNO) compare spessissimo, quasi sempre.  
Questo però implica la necessità di una gerarchia per gli operatori, in modo che quando due di essi, coabitando la stessa espressione, dovessero legittimamente contendersi un operando, si possa dirimere quale dei due se lo aggiudica.  
Legittimamente contendersi non è una terminologia oziosa: significa che se si scrive un codice in cui, manifestamente, due operatori entrano tra loro in lizza senza alcun diritto, ci pensa il compilatore a segnalare errore; al contrario, se la contesa è legittima, il compilatore deve poter assumere la funzione di arbitro e assegnare l'operando a uno dei due operatori secondo regole convenute: è qui che entrano in gioco i concetti di *precedenza* e di *associatività*.  
  
Tanto per intendersi, nelle due espressioni seguenti, in cui ogni variabile sia dichiarata come *int*:

z = x + y \* w;  
z = & x + y \* w;

la prima presenta una legittima contesa tra l'operatore *\** e l'operatore *+* a carico dell'operando *y*, mentre la seconda contiene una contesa illegittima tra l'operatore *+* e l'operatore unario *&* a carico dell'operando *x* (e infatti il compilatore la risolve segnalando l'errore e NON compilando il programma).  
  
Le regole per risolvere le contese legittime sono molto spesso ovvie e senza sorprese; ad esempio, nella prima delle due espressioni sopra riportate, dovrebbe essere palese a tutti che *y* debba essere l'operando sinistro di *\** e NON l'operando destro di *+* che avrà bensì come operando destro il risultato della moltiplicazione (come tale costituito da un valore temporaneo, privo di indirizzo in memoria; **RICORDATEVI QUESTO FATTO**). E così arbitrerà, come ci si aspetterebbe, il compilatore, fermo restando che, se l'arbitrato del compilatore non è soddisfacente, si possono sempre usare le coppie di parentesi per alterarlo scrivendo *z = (x+y)\*w;*.  
  
Tuttavia ci sono dei casi in cui le decisioni che il compilatore prenderà non sono immediatamente chiare, se non dopo aver acquisito una lunga esperienza ed essersi letto con cura QUESTO canto. Ad esempio, il compilatore **SA BENISSIMO** come risolvere la seguente contesa sull'operando *p*, legittimamente dichiarato come puntatore a *int* (o a qualsiasi altro tipo) e altrettanto legittimamente inizializzato sì da puntare un certo numero di elementi (oggetti del proprio tipo):

\*  p  ++;

ma secondo voi la risolverà in quale dei seguenti modi?

1. Si tratta di un errore: la contesa è illegittima.
2. Si tratta di incrementare di 1 il valore intero del primo elemento puntato dal puntatore *p*, perché è l'operatore di dereferenza (l'asterisco unario) a vincere la contesa.
3. Si tratta di incrementare di 1 il puntatore *p* stesso, perché è l'operatore di incremento *++* a vincere la contesa.

Non è così ovvio, vero?  
  
La risposta corretta è la numero 3., per cui a trovarsi incrementato di 1 è il puntatore *p*, mentre l'operatore *\** accede COMUNQUE al PRIMO oggetto contenuto nell'area di memoria puntata da *p*, a causa della cronologia di applicazione degli effetti collaterali, come sarà diffusamente discusso e spiegato nel prossimo canto: nessuno degli oggetti puntati subisce incrementi di sorta.  
  
La seguente tabella è la ricetta utilizzata dal compilatore per la risoluzione di ogni **legittima** controversia tra operatori, nel senso che si è cercato di spiegare fin qui, fermo restando che qualsiasi controversia illegittima provoca ERRORE (l'illegittimità di un conflitto tra operatori dovrebbe essere facilmente implicata dall'unione proficua tra la definizione dell'operatore e le virtù del programmatore): ogni volta che due operatori sono in concorrenza su un operando, la spunta quello dei due che occupa una casella di indice più basso nella tabella.

|  |
| --- |
| 1 |
| :: |
| 2 |
| ++   -- (suffissi) | (    )  chiamata di funzione | [  ] | .    -> |
| 3 |  |  |  |
| ++   -- (prefissi) | +   - (unari) | !   ~ | \*   &(unari) | sizeof new delete | (casting) |
| 4 |  |  |  |  |  |
| .\*   ->\* |  |  |  |  |  |
| 5 |  |  |  |  |  |
| \* (binario)  /  % |  |  |  |  |  |
| 6 |  |  |  |  |  |
| +   -   (binari) |  |  |  |  |  |
| 7 |  |  |  |  |  |
| <<   >> |  |  |  |  |  |
| 8 |  |  |  |  |  |
| <   >   <=   >= |  |  |  |  |  |
| 9 |  |  |  |  |  |
| ==   != |  |  |  |  |  |
| 10 |  |  |  |  |  |
| &  (binario) |  |  |  |  |  |
| 11 |  |  |  |  |  |
| ^ |  |  |  |  |  |
| 12 |  |  |  |  |  |
| | |  |  |  |  |  |
| 13 |  |  |  |  |  |
| && |  |  |  |  |  |
| 14 |  |  |  |  |  |
| || |  |  |  |  |  |
| 15 |  |  |  |  |  |
| ?:   ogni operatore di assegnamento |  |  |  |  |  |
| 16 |  |  |  |  |  |
| throw |  |  |  |  |  |
| 17 |  |  |  |  |  |
| ,    (virgola) |  |  |  |  |  |

Gli operatori che non figurano nella tabella (cfr. precedente canto) non sono portati a conflitti con altri operatori.  
  
In base alla tabella risulta spiegata la risoluzione del conflitto nel caso dell'espressione *\*p++*; in effetti l'asterisco unario è inserito nella celletta numero 3 e quindi perde nei confronti del *++* suffisso, che occupa la celletta 2. Pertanto se l'intenzione del programmatore fosse stata quella di far aderire il proprio codice alla risposta 2 citata sopra, avrebbe dovuto ricorrere alle parentesi scrivendo *(\*p)++*.  
  
Che succede però quando a concorrere sullo stesso operando sono due operatori appartenenti alla STESSA cella?  
  
Qui il compilatore ricorre al concetto di *associatività* che può essere di due tipi: o da sinistra a destra o da destra a sinistra. Queste locuzioni significano semplicemente in quale verso della sorgente di programma sono elaborate le espressioni in cui compaiono operatori di uguale precedenza (nel senso della tabella) in concorrenza sullo stesso operando.  
  
Ogni cella della tabella è caratterizzata da una ben determinata forma di associatività: la maggior parte ha l'associatività *da sinistra a destra*, mentre quella *da destra a sinistra* è propria SOLTANTO delle celle numero 3 e numero 15. Ad esempio, che ne è dell'espressione *++\*p*, ove *p* è sempre lo stesso puntatore di prima? Stavolta i due operatori coinvolti sono di uguale precedenza, appartenendo entrambi alla cella 3; ma poiché l'associatività di quella cella, come appena detto, è *da destra a sinistra*, l'espressione si risolve, come appare anche sensato, nell'incremento di 1 del primo valore puntato dal puntatore. Altrettanto logicamente, invece, *\*++p* è il valore del SECONDO elemento puntato da *p* (ammettendo che **CI SIA**), che è LUI a essere stato incrementato. **PROVATE PER CREDERE**.  
  
Il concetto di associatività ha rilevanza ANCHE QUANDO SEMBREREBBE CHE NON DEBBA AVERLA. Considerate la seguente espressione, ove le variabili coinvolte sono dichiarate come vi pare, col solo vincolo di mantenere valutabile l'espressione stessa (**METTETEVI IN TESTA QUESTA FRASE, NONOSTANTE POSSA SEMBRARVI OZIOSA QUANDO LEGGERETE L'ESPRESSIONE**):

z = t + u + v;

Qui *u* è operando sinistro del secondo operatore *+* o operando destro del primo? La risposta che sembrerebbe più naturale sarebbe *E CHI SE NE FREGA?* (confidando sulle auree proprietà commutativa e associativa dell'addizione). Tuttavia, per le regole del compilatore, la risposta giusta è la seconda e questo NON È IRRILEVANTE perché potrebbe anche succedervi che la proprietà commutativa dell'addizione VE LA DOBBIATE SCORDARE e che l'espressione di sopra sia GIUSTA (ossia compilabile), mentre quella ottenuta scambiando di posto tra loro due variabili qualsiasi **NON LO SIA PIÙ**.

**Canto quattordicesimo: operatori, precedenze, associatività, sequenzialità e *pietre miliari*; parte III: sequenzialità e *pietre miliari***

La lettura accurata dei precedenti due canti vi dovrebbe aver resi persuasi che, di fronte a un'espressione *la più complicata che si possa ipotizzare* il compilatore segue precise e univoche regole, dettate dallo standard del linguaggio e per ciò stesso conoscibili anche dal (buon) programmatore, per attribuire quale sottoespressione come operando a quale operatore (rileggete a fondo i due canti citati, se vi restano dei dubbi).  
  
Tuttavia lo standard del linguaggio NON DICE QUASI NULLA a proposito della cronologia di esecuzione delle valutazioni di ogni sottoespressione all'interno di un'espressione composita; ciò significa che se il buon funzionamento di un programma fa confidenza su tale cronologia, QUEL PROGRAMMA È **SCRITTO MALE**, perché potrebbe avere comportamenti diversi su diversi calcolatori e **ADDIRITTURA** sullo stesso calcolatore in esecuzioni diverse.  
  
Ciò dipende dal fatto che, sulle moderne architetture dei calcolatori, i processori sono in grado di svolgere diverse azioni SIMULTANEAMENTE e se due azioni (chiamiamole **A** e **B**) sono portate avanti contemporaneamente dal processore, al programmatore non può essere dato di prevedere, in generale, quale si concluda per prima: pertanto, se la buona riuscita del programma si fondasse sul fatto che, quando si intraprenda l'azione **B**, l'azione **A** sia già conclusa poiché i suoi esiti servono al compimento dell'azione **B**...evidentemente non ci si può aspettare che il programma funzioni a dovere.  
  
Come ben si dovrebbe intuire, qui si trattano problematiche piuttosto fini e subdole, che meritano assolutamente di essere **BEN COMPRESE** da chi voglia conseguire il titolo onorifico di **buon programmatore C++**.  
Tuttavia non bisogna spaventarsi, dato che sopra è scritto *QUASI NULLA*, **non** *NULLA AFFATTO*, il che significa che qualcosa da imparare comunque C'È.  
  
In effetti lo *standard* del linguaggio contempla i concetti di *sequenzialità* e di quelle che io ho tradotto come pietre miliari (in angloamericano *sequence points*), ossia precisi e *standardizzati* luoghi, all'interno del codice, in cui il programmatore può metaforicamente sedersi su un cippo al lato della strada e fermarsi a considerare quanta ne sia già stata fatta e quanta ne resti ancora da fare.  
  
Al di là delle immagini poetiche da cui talvolta lascio prendermi la mano, occorre innanzitutto ricordare quali siano le azioni che il compilatore fa compiere alle CPU (*Central Processing Units*, in angloamericano) quando traduce in codice binario le espressioni scritte dal programmatore, così, tanto per capire di che cosa si sta parlando. Giova pertanto ripetere ancora una volta che si tratta di DUE TIPI di azioni:

1. determinazione del **valore** di una (sotto)espressione;
2. applicazione degli **effetti collaterali** implicati da una (sotto)espressione.

Il concetto di *sequenzialità* previsto nello *standard* del linguaggio recita sostanzialmente che esistono azioni che DEVONO essere COMPLETATE PRIMA che altre abbiano inizio, e questa è senz'altro una gran buona notizia. Il compilatore si premurerà di riconoscerle e di inserirle nel codice binario in modo tale da rispettarne la cronologia esecutiva.  
D'altra parte, però, esistono anche azioni che **NON NECESSITANO** di un'esecuzione in rigoroso ordine cronologico una rispetto all'altra; davanti a queste il compilatore È LASCIATO LIBERO DI AGIRE COME MEGLIO CREDE scegliendo **LUI** una tra queste tre modalità (usiamo ancora **A** e **B** come nome per le due azioni in questione):

* **O** produrre un codice binario in cui l'azione **A** sia compiuta ed esaurita PRIMA che l'azione **B** abbia inizio (si dice che **A** è *presequenziata* a **B** o, equivalentemente, che **B** è *postsequenziata* ad **A**);
* **O** produrre un codice binario in cui l'azione **B** sia compiuta ed esaurita PRIMA che l'azione **A** abbia inizio (si dice il contrario di quanto detto sopra);
* **O** produrre un codice binario in cui le due azioni siano portate avanti **contemporaneamente** (si dice che **A** e **B** sono *asequenziate*).

Come il compilatore compia le proprie scelte dipende sia dall'architettura della macchina su cui agisce sia dalle contingenti condizioni di lavoro della macchina stessa: in ogni caso si comprende bene (spero) che, a partire dalla stessa fonte di codice, possono prodursi codifiche binarie completamente diverse ANCHE SULLA STESSA MACCHINA se, ad esempio, si compila il programma di giovedì piuttosto che di domenica... ed è per questa ragione che un programma **NON PUÒ BASARSI** su questo tipo di congiunture astrali se vuol essere un programma serio.  
  
Al programmatore non resta quindi che apprendere e mandare a menadito quali azioni, secondo lo *standard*, godano di *sequenzialità* chiara e univoca, e regolarsi di conseguenza. Ecco un *vademecum*, spero esauriente:

1. c'è una pietra miliare a OGNI punto e virgola, il che significa che TUTTE le azioni, di qualunque genere siano, implicate nella valutazione di un'INTERA ESPRESSIONE sono *presequenziate* rispetto a qualunque azione, di qualunque genere, occorra a valutare l'espressione successiva (ALMENO QUESTO È SICURO...);
2. l'applicazione dell'effetto collaterale (azione di tipo 2.) degli operatori di incremento e decremento **suffissi** è *postsequenziata* rispetto alla determinazione del valore del loro operando (azione di tipo 1.).
3. l'applicazione dell'effetto collaterale (azione di tipo 2.) degli operatori di incremento e decremento **prefissi** è *presequenziata* rispetto alla determinazione del valore del loro operando (azione di tipo 1.).  
   Cumulando i tre punti fin qui proposti si ricava che in   
   *int i = 0;   
   cout << ++i <<'\n';// pietra miliare: i incrementato  
   cout << i++ <<'\n';// pietra miliare: i incrementato  
   cout << i <<'\n';*   
   si vedranno apparire, uno per riga, i valori 1, 1, 2;
4. **tutte** le azioni, di qualsiasi genere, coinvolte nella valutazione del VALORE dell'operando SINISTRO dei tre operatori *&&*, *||* e *,* (virgola) sono *presequenziate* rispetto a qualsiasi azione coinvolta nella valutazione del loro operando DESTRO. In altre parole c'è una pietra miliare in corrispondenza di ciascuno di questi operatori, per cui, almeno per quanto riguarda la sequenzialità delle azioni, la virgola si comporta come il punto e virgola (salvo il suo essere un operatore, che la distingue nettamente; confrontare anche quanto detto al successivo punto 7.);
5. **tutte** le azioni, di qualsiasi genere, coinvolte nella valutazione della prima delle tre espressioni previste nella sintassi dell'operatore ternario condizionale (punto interrogativo) sono *presequenziate* rispetto a qualsiasi azione coinvolta nella determinazione delle altre due espressioni. In altre parole c'è una pietra miliare in corrispondenza del punto interrogativo;
6. ogni azione coinvolta in qualsiasi espressione contenuta nell'ambito di una funzione è *postsequenziata* SIA rispetto a **tutte** le azioni, di qualsiasi genere, coinvolte nella determinazione dei valori dei parametri da trasferire alla funzione stessa all'atto della sua invocazione, SIA rispetto a **tutte** le azioni, di qualsiasi genere, coinvolte nella determinazione del valore di un'eventuale espressione postfissa all'invocazione della funzione. Significa, ad esempio, che nel seguente segmento di codice  
   *int x = 4, y = 4;  
   funz(x-y, x+y);*possiamo essere CERTI che la funzione NON inizia senza che siano già stati attribuiti i valori 0 e 8 ai suoi argomenti; altrettanto accade se c'è un'espressione postfissa come in  
   *funza(x, y) -> g(x-y, x+y);*
7. Nella valutazione degli operandi di **qualsiasi** operatore è *presequenziata* rispetto alla determinazione del risultato dell'operazione SOLO la valutazione del valore di ciascun operando (azione di tipo 1.), **NON** l'applicazione di eventuali effetti collaterali (azione di tipo 2.). Fa eccezione la virgola, per la quale, a conferma di quanto asserito al precedente punto 4., è presequenziata anche l'applicazione degli effetti collaterali. Significa che in  
   *int i = 3, x;  
   x = i++ + 2;*possiamo essere CERTI che l'operatore binario di addizione opera su operandi di valore 3 e 2 rispettivamente, ma NON ABBIAMO NESSUNA CERTEZZA sul fatto che l'incremento di *i* (che è *postsequenziato* rispetto alla determinazione del suo valore, come detto al punto 2 di QUESTO elenco) avvenga prima, dopo o insieme con il calcolo del risultato dell'addizione. Sappiamo solo che è CERTAMENTE compiuto al punto e virgola (punto 1 di questo elenco) o anche, se presente, alla virgola che lo sostituisca lecitamente (punto 4.);
8. Quando un'espressione contiene una versione qualsiasi degli operatori di assegnamento (vale a dire quasi tutte) l'effetto collaterale dell'operatore, ossia la **modifica** del suo operando SINISTRO, è *postsequenziato* rispetto alla determinazione del valore dei due operandi, ma NON degli effetti collaterali connessi (esattamente come per qualsiasi altro operatore, ex punto precedente), MA *PRESEQUENZIATO* rispetto alla valutazione del valore dell'intera espressione. Significa che, ad esempio, in:  
   *int i, x, z = 4, y = 1;  
   i = x = z += y++;*sia *i* sia *x* assumono il valore assunto **ANCHE** da *z* DOPO che gli è stato aggiunto 1. Nulla è dato conoscere sull'istante relativo in cui viene incrementato *y*.  
   Si sottolinea che, quando è coinvolto l'operatore di assegnamento, la cosiddetta valutazione dell'operando SINISTRO (*presequenziata*, come detto) consiste più propriamente nella determinazione (*presequenziata*) di QUALE SIA la locazione in memoria cui applicare l'effetto collaterale (*postsequenziato*).
9. in una lista di inizializzatori racchiusa tra parentesi graffe, la virgola che separa ciascun inizializzatore dal successivo agisce, sotto il profilo della *sequenzialità*, ESATTAMENTE come detto al punto 4. di QUESTO stesso elenco. Significa che in  
   *int i=10, a[  ]{i++, i};*  
   possiamo con CERTEZZA ritenere inizializzato il contenuto dell'array *a* coi valori 10 e 11;
10. se le esecuzioni di due funzioni NON sono in alcun modo obbligatoriamente sequenziate una rispetto all'altra (come avviene, ad esempio, quando i loro risultati sono operandi di un operatore binario: *z = funza( ) + funzb( )*, ex punto 7. di QUESTO elenco), tra le tre possibili scelte a disposizione del compilatore è esclusa la terza; ciò significa che comunque UNA DELLE DUE funzioni è eseguita e completata PRIMA che l'altra abbia anche solo inizio, MA NON SI PUÒ SAPERE QUALE DELLE DUE;
11. quest'ultimo punto è prematuro, ma viene ugualmente citato per salvaguardia della completezza: NON vi è alcuna sequenziazione certa tra l'allocazione della memoria operata dall'operatore *new* e la valutazione di parametri da trasmettere a un eventuale costruttore che inizializzi quella memoria...(RICORDATEVELO A TEMPO DEBITO...)

Fin qui ciò di cui si può essere SICURI; vediamo ora di puntualizzare meglio ciò di cui invece non si può essere SICURI affatto e cui si è già accennato, in parte, tra le righe stesse dell'elenco appena presentato.  
Ad esempio, quando nel punto 6. si dice quanto colà detto, è forse opportuno sottolineare che non vi è ALCUNA SEQUENZIAZIONE OBBLIGATA nella valutazione dei parametri da trasferire a una funzione l'UNO RISPETTO ALL'ALTRO, ma solo quella di TUTTI i parametri rispetto all'esecuzione della funzione. Significa che, restando VERO e VALIDO l'esempio riportato al punto 6., nel seguente **controesempio**:  
*int x = 1;  
funz(x, x++);*non potremmo essere in alcun modo certi che la funzione riceva come argomenti la coppia 1,1 o la coppia 2,1; in altri termini, nella lista dei parametri per l'invocazione di una funzione, la virgola NON AGISCE come nei punti 4. e 9. dell'elenco.  
E quantunque viga il punto 3., nemmeno in questa situazione:  
*int x = 1;  
funz(++x, ++x);*si potrebbe sapere se la funzione riceve la coppia 2,3 o la coppia 3,2. E per quanto si è detto a proposito dell'operatore di assegnamento al punto 8., perfino in una situazione apparentemente lampante come:  
*int x;  
funz(x = 1, x = -1);*NON si potrebbe essere sicuri di che cosa perviene alla funzione e, a dirla tutta, NEPPURE DI QUALE SIA IL VALORE con cui la variabile *x* ne esce, con gli esiti catastrofici che potrebbero derivare da ciascun controesempio.  
  
Altresì non è possibile avere ALCUNA CERTEZZA in situazioni come questa:  
*int i = 2;  
i = i++ + 1;*o quest'altra:  
*int i = 0, a[2];  
a[i] = i++;*sempre a causa delle sequenzialità non esplicitamente previste dall'operatore di assegnamento.  
  
E finiamo con quest'altra situazione, in cui non si può essere certi di che cosa apparirà in *output*, perché la valutazione presequenziata degli operandi di ogni operatore (ex punto 7.) NON IMPLICA ALCUNA SEQUENZIAZIONE della valutazione di un operando rispetto all'altro quando c'è di mezzo un operatore binario e per giunta ci sono effetti collaterali pendenti come la spada di Damocle.  
*int i = 0;  
cout << i << ' ' << i++ <<'\n';*  
**EVITATE QUINDI SITUAZIONI DI NON SICURA SEQUENZIAZIONE SÌ COME SI EVITANO L'AIDS E IL COLERA**.  
  
Ogni lettore attento avrà percepito (forse) che in alcune voci dell'elenco puntato si è parlato di impossibilità di conoscere il momento esatto in cui avviene una certa azione; il più delle volte, peraltro, questa non conoscenza non appariva cruciale. Ad esempio, nel punto 7. ci si diceva ignoranti su quando *i* avrebbe subìto incremento: ma nell'espressione colà scritta ciò appare del tutto irrilevante, visto che SUBITO DOPO c'è il punto e virgola...  
Verissimo, ma per questa sensata e attenta obiezione ci sono subito pronte due controdeduzioni:

1. innanzitutto non è detto che il punto e virgola, o perfino la semplice virgola, siano sempre lì, immediatamente pronti a cavare le castagne dal fuoco: immaginate una (sotto)espressione che prosegue per cinque o sei righe con qualche dozzina di altri operatori coinvolti...
2. in secondo luogo, e in modo **ANCOR PIÙ DECISIVO**, non va dimenticato che in questa pagina si è sempre dato per sottinteso che gli operatori di cui si parlava erano quelli che il linguaggio prevede... ...ma perché, CE NE SONO ALTRI??? (questa domanda vi sarebbe dovuta sgorgare dall'interno delle vostre frattaglie)  
   **E COME SE CE NE SONO ALTRI...** praticamente in numero infinito; tutti quelli che il buon programmatore può CREARE per farli agire sui tipi che egli stesso pure CREA. Va da sé, a questo punto, che una (quasi) PERFETTA conoscenza e PADRONANZA di TUTTI gli ultimi tre canti che avete letto sia ALTISSIMAMENTE AUSPICABILE.

Ah, un'ultima cosa di ESTREMA RILEVANZA: per quanto detto al punto 4. e per la natura booleana del risultato prodotto dagli operatori *&&* e *||*, il fatto che sia posta una pietra miliare subito dopo la completa valutazione del loro operando sinistro potrebbe implicare che l'operando destro **SIA TOTALMENTE IGNORATO**, né più né meno come se neppure ci fosse. Ciò accade quando la valutazione del primo operando dà il risultato **false** per l'operatore *&&* e **true** per l'operatore *||*. In effetti, in ciascuno di questi due casi, il valore finale dell'espressione è già conseguito, SENZA BISOGNO del concorso dell'operando destro.  
Per intendersi, in entrambe le espressioni *true || funz(  );* e *false && funz(  );* la funzione *funz(  )* **NON È ESEGUITA AFFATTO**.  
Lo stesso accade, coerentemente, a quella delle due ultime espressioni costituenti il secondo e terzo operando dell'operatore condizionale ternario che non corrispondesse al valore booleano del primo operando, completamente valutato alla pietra miliare posta sul punto interrogativo. In:  
*true    ?    funz(   ) :   x = gunz(   );*   
né la funzione *gunz(   )* è eseguita né, *a fortiori*, la variabile *x* è inizializzata, mentre in:  
*false    ?    funz(   ) :   x = gunz(   );*   
a non essere affatto eseguita è *funz(   )*, e *x* è inizializzata col valore restituito da *gunz(   )*.  
[Spero che sia chiaro a tutti che questi ultimi sono solo esempi, e che usare l'operatore condizionale ternario, o gli operatori *&&* e *||*, ponendo una costante esplicita come primo operando è peggio della *"corazzata Potëmkin"* secondo Fantozzi...]

**Canto quindicesimo: come aspettarsi l'inaspettato; la *cattura delle eccezioni***

Cominceremo questo canto con un altro assioma da mandare a mente:  
  
**IL BUON PROGRAMMATORE DEVE PENSARLE TUTTE E TENDERE A RIDURRE A ZERO IL NUMERO DEGLI EVENTI INATTESI**   
  
In altre parole il buon programmatore non dovrebbe mai lasciarsi cogliere di sorpresa da ciò che potrebbe accadere durante l'esecuzione del proprio programma, a meno che la sua non sia una scelta volontaria, dovuta, per esempio, al rifiuto di far eseguire i propri codici da parte di *minus habentes* sul piano dell'intelletto (scimmie catarrine o psittaciformi di varie specie).  
  
Il linguaggio viene incontro al programmatore fornendo ben quattro parole di vocabolario dedicate alla cosiddetta gestione (o cattura, alla fin fine) delle eccezioni ossia di quegli imprevisti che, una volta gestiti e catturati, cessano di essere tali.  
  
Le parole in questione sono:

* *try*
* *catch*
* *throw*
* *noexcept*

Le prime due sono legate da un rapporto coniugale simile, ma NON uguale, a quello instaurato fra *if* ed *else*, le differenze essendo:

* *try* è PER FORZA coniugato, mentre *if* poteva rimanere celibe;
* a *try* è consentita la poligamia, mentre *if*, se coniugato, resta rigorosamente monogamo.

In ogni caso, come *if* precede sempre il suo (eventuale) unico coniuge *else*, anche *try* precede sempre TUTTI i suoi (eventualmente molteplici, ma almeno UNO obbligato) coniugi *catch*. In caso di *try* poligamo, l'ordine relativo con cui sono inseriti nel codice i diversi *catch* HA RILEVANZA SOLO IN DETERMINATI CASI che saranno appresso specificati; in genere basta solo, appunto, che siano TUTTI DIVERSI, nel senso che or ora sarà precisato, e uno CONSECUTIVO all'ALTRO, vale a dire che, in mezzo a due *catch* che siano entrambi coniugi dello stesso *try*, non deve essere frapposto nemmeno il più minuscolo segmento di codice estraneo, **NEMMENO UN MISERRIMO** punto e virgola orfano di espressione.  
  
La terza parola e la quarta (*throw* e *noexcept*) sono trattate dal compilatore come operatori e hanno già fatto una fugace apparizione in qualcuno dei canti precedenti, dove si è anche introdotto, senza spiegazioni, il concetto di *throw expression* (ANDATE A RICERCARVELE).  
  
È giunta l'ora di presentare la/le sintassi dovute:

**try**  
{   
// APERTURA DI AMBITO: **OBBLIGATORIA**  
// **ANCHE IN CASO DI SINGOLA ESPRESSIONE**  
// quante si vogliano linee di codice di ogni genere  
}   
// FINE dell'ambito di *try*; perdita di OGNI  
// variabile eventualmente dichiarata al suo interno  
  
**catch(int z)** // catch **OBBLIGATORIO**  
{   
// APERTURA DI AMBITO: **OBBLIGATORIA** c. s.  
// quante si vogliano linee di codice di ogni genere  
}   
// FINE dell'ambito di *catch(int)*  
  
// ; // QUESTO punto e virgola **DAREBBE ERRORE**  
  
**catch(long int z)** // catch **FACOLTATIVO**  
{   
// APERTURA DI AMBITO: **OBBLIGATORIA** c. s.  
// quante si vogliano linee di codice di ogni genere  
}   
// FINE dell'ambito di *catch(long int)*  
// QUANTI ALTRI catch (FACOLTATIVI) si vogliano,  
// OGNUNO PERÒ CON UNA DIVERSA   
// DICHIARAZIONE IN PARENTESI  
// L'**ULTIMO** catch inserito può, **OPZIONALMENTE**,  
// avere tre puntini al posto della dichiarazione,  
// ossia essere scritto come *catch(...)*  
// (è anche ammissibile che un *catch* così scritto  
// sia l'unico, **OBBLIGATORIO**, presente)

Il significato delle sintassi puntigliosamente descritte dagli stessi commenti inseriti è il seguente:  
Quando il programma incontra la parola *try* prosegue l'esecuzione come se niente fosse, semplicemente entrando nel suo ambito; Se, per tutto il tempo in cui l'esecuzione permane nell'ambito di *try*, ivi comprese le istruzioni contenute in qualsivoglia funzione eventualmente invocata, e **fino a qualsiasi livello di nidificazione di invocazioni**, NON viene mai eseguito l'operatore *throw*, allora, all'uscita dall'ambito di *try*, TUTTE le espressioni contenute in TUTTI gli ambiti *catch* relativi a quel *try* **VENGONO SALTATE A PIE' PARI** e l'esecuzione continua a partire dal punto immediatamente successivo all'ultima graffa chiusa dell'ultimo *catch*.  
  
Viceversa, **APPENA** venisse eseguito l'operatore *throw*, **NON IMPORTA DA CHI E QUANDO** (nel senso che potrebbe farlo anche la funzione *funzZ*, invocata da *funzY*, invocata da *funzX*, ... invocata da *funzA* invocata nell'ambito di *try*), l'esecuzione del programma verrebbe ISTANTANEAMENTE INTERROTTA in QUEL PUNTO e riprenderebbe ENTRANDO NEL PRIMO AMBITO *catch* (tra quelli collegati al *try*) contrassegnato da una dichiarazione di tipo COMPATIBILE con quello dell'espressione su cui ha operato *throw* (appunto denominata *throw expression*).  
Se un *catch* cotale NON ESISTE (colpa o volontà del programmatore), MA ESISTE il *catch(...)* (quello coi tre puntini), **SARÀ QUEST'ULTIMO** a prendere il controllo del programma; se NON ESISTE NEMMENO il *catch(...)* l'azione dell'operatore *throw* sarà l'ultima intrapresa dal programma sotto controllo da parte del programmatore, perché, in tale caso, il programma si concluderà DEFINITIVAMENTE eseguendo la funzione di sistema *abort(  )*.  
  
In ogni caso, se il programma riesce a entrare in uno dei *catch* forniti dal programmatore, TUTTI GLI ALTRI (eventuali) SARANNO ESCLUSI DALL'ESECUZIONE, anche se scritti nel codice DOPO quello innescato, e, all'uscita dal *catch* chiamato in causa (se il programmatore non dispone diversamente, ad esempio terminando volontariamente l'esecuzione dall'interno del *catch*, **OVVERO eseguendovi novamente l'operatore *throw***) l'esecuzione continua riprendendo dal punto immediatamente successivo a TUTTI i *catch*.  
  
Si fa prima a farlo che a dirlo; spero tuttavia che si sia compresa la portata e la produttività di questa facoltà fornita dal linguaggio al buon programmatore, ossia poter trasferire ISTANTANEAMENTE il controllo del programma da una funzione a un'altra che ne dista topologicamente nello *stack* delle esecuzioni delle funzioni un numero arbitrario di passi, realizzando in un battito di ciglia il cosiddetto *stack unwinding* ossia, in pratica, l'esecuzione di tutti i *return* delle funzioni invocate.  
  
L'esecuzione dell'operatore *throw* avviene nella forma intuitiva

throw espressione;

ove *espressione* è, appunto, qualsiasi espressione NON di tipo *void*. Quando sopra si è detto che il tipo cui appartiene *espressione* DECIDE se e in quale *catch* far entrare il programma, e si è aggiunto che quel *catch* sarà il PRIMO che presenta una dichiarazione compatibile (**NON NECESSARIAMENTE COINCIDENTE DEL TUTTO**) col tipo di *espressione*, si è detto quel che serve su quanto rilevi l'ORDINE con cui i *catch* sono inseriti nel codice: ecco perché il *catch* coi tre puntini, se c'è, VA INSERITO per ULTIMO (essendo compatibile con QUALSIASI tipo); del resto il compilatore si fa carico di segnalare errore se non si ottempera a quest'ovvia restrizione.   
  
Il seguente codice, eseguibile così com'è, non solo esemplifica quanto fin qui detto, ma presenta anche alcune sintassi alternative per le parole *try* e *throw*, oltre a introdurre anche l'operatore *noexcept*:

# include <iostream>  
# include <cmath>  
# include <cstdlib>  
using namespace std;  
  
int int\_funz( ) noexcept  
{return 10;}  
  
double double\_funz( ) noexcept(true)  
{return 11.;}  
  
void funz(int k) throw(int, double)  
{switch(k)  
{case 0: throw int\_funz( );  
case 1: throw double\_funz( );}}  
  
void gunz(double n) throw(int, double)  
{if(n > 0)  
cout   
<< "log(" << n   
<< ") = " << log(n) << '\n';  
else if(n < 0)  
cout   
<< "non puoi fare logaritmi di numeri negativi, suvvìa!\n", funz(1);  
else cout << "passiamo oltre ... \n", funz(0);}  
  
void hunz(int u) try   
{gunz(u);}   
catch(...)  
{cout << "sono catch(...) in hunz\n"; throw;}  
  
int main( )  
{while(true)  
{try   
{double x; cout << "dimmi un numero ",  
cin >> x, gunz(x);}  
catch(int x) {continue;}  
catch(double x) {break;}}  
int n;  
cout << "adesso dimmi un intero qualsiasi ", cin >> n;  
try {hunz(n);}   
catch(...)   
{cout << "sono catch(...) in main\n", exit(22);}  
cout  
<< "se vedi questo hai digitato un intero positivo!\nBuona giornata\n";}

Il programma calcola logaritmi naturali finché si continua a fornirgli numeri positivi (la funzione *log*, assieme a tutte le altre funzioni matematiche elementari, è dichiarata nel documento *cmath* incluso; invece il documento *cstdlib* deve essere incluso per poter usare la funzione *exit*); se si fornisce zero il programma lo ignora e prosegue regolarmente il ciclo di calcolo (viene eseguito *continue*), mentre se si fornisce un numero negativo il ciclo si interrompe con un monito (viene eseguito *break*).  
All'uscita dal ciclo il programma richiede un ultimo dato e riesegue le stesse valutazioni/operazioni, stavolta però tramite una funzione (*hunz*) che fa da ulteriore involucro a quella che svolge le operazioni vere e proprie: *gunz*, che, a sua volta, fa da involucro a *funz* per quanto riguarda l'esecuzione di *throw*.  
  
**ESEGUITE IL PROGRAMMA E RISPONDETEGLI COME VI GARBA, CERCANDO DI CAPIRNE IL FUNZIONAMENTO PRIMA DI LEGGERE LE SPIEGAZIONI DETTAGLIATE CHE SEGUONO. DATE RETTA, CHÉ VI CONVIENE.**  
  
Il programma si dispone a *eseguire eternamente*, a causa di *while(true)*, un segmento di codice sottoposto alla gestione di *try*, nel cui ambito ci si limita a richiedere la lettura di un dato il cui valore è trasmesso a una funzione *gunz* che viene eseguita: **NON VI È TRACCIA DI *throw*** visibile in tale segmento.  
  
La funzione *gunz*, la cui esecuzione è stata appunto richiesta dall'interno dell'ambito di *try*, è marcata, all'atto della sua dichiarazione/definizione, con la clausola **OPZIONALE** *throw(int, double)*, nella posizione in cui si trova. Significa che, per quanto la riguarda, qualsiasi operatore *throw* che fosse eseguito da parte di codice che trae origine dal suo interno, è autorizzato ad agire con espressioni appartenenti SOLO ai tipi indicati; tuttavia NEMMENO *gunz* esegue mai direttamente l'operatore *throw*, limitandosi a calcolare e scrivere il valore del logaritmo del proprio argomento, quando è positivo, ovvero a invocare la funzione *funz*, trasmettendole valori diversi, quando l'argomento ricevuto fosse negativo o nullo (e facendone precedere l'esecuzione dalla scrittura di messaggi pittoreschi).  
  
A sua volta *funz*, marcata come *gunz*, qualora venisse invocata, usa il proprio argomento per eseguire finalmente l'operatore *throw*, **DA DUE LIVELLI DI CHIAMATE DI FUNZIONI PIÙ IN BASSO RISPETTO ALLA COLLOCAZIONE DEL *try***, con due espressioni di tipo diverso, secondo il valore ricevuto, ma comunque elencato nella clausola *throw(int, double)*. I valori delle *throw expressions* sono forniti dalle due semplicissime funzioni *int\_funz* e *double\_funz*, marcate entrambe con l'operatore *noxcept* in due modi apparentemente diversi, ma funzionalmente equivalenti, col significato che, dal codice originato dall'esecuzione di funzioni così marcate, non è prevista l'esecuzione di alcun operatore *throw* (in queste due forme equivale anche a *throw(  )*; dettagli più fini sono rinviati a tempo debito).  
  
Appena *funz* esegue una qualsiasi delle sue due espressioni *throw* il controllo di esecuzione risale indietro TUTTI I LIVELLI DI INVOCAZIONE DI FUNZIONI PENDENTI (nel caso presente DUE LIVELLI) liberando a ogni gradino risalito tutta la memoria impegnata per variabili create negli ambiti delle funzioni coinvolte (tale processo viene appunto chiamato, tecnicamente, *stack unwinding*), e IMBOCCA l'appropriato *catch* portandosi dietro il valore della *throw expression* nella variabile dichiarata nell'ambito di *catch* stesso: nel caso in esame la variabile *x*, adeguatamente dichiarata in ciascuno dei due *catch*, e, volendo, disponibile nell'ambito (provate, per esempio, ad aggiungervi un'istruzione di scrittura). Secondo quale *catch* è innescato, l'esecuzione, in alternativa, di *continue* o di *break* spiega come si comporta il codice prima di scrivere il messaggio *"adesso dimmi un intero qualsiasi"* (**SI SUPPONE CHE LO STIATE ESEGUENDO!**).   
  
Quel che viene digitato in risposta, è trasmesso come parametro alla funzione *hunz*, invocata dall'interno di un nuovo *try*; si osservi che questa funzione ha il suo intero ambito anch'esso sottoposto a *try* (UN ALTRO *try*, DENTRO quello esterno posto in *main*): tale è il significato di questa parola quando si trova nella posizione riportata nella definizione di *hunz*. Allora, coerentemente con la sintassi, DEVE ESSERCI ALMENO UN *catch* CHE SEGUE QUEL *try* e che, quindi, non appartiene all'ambito di ALCUNA FUNZIONE.  
  
Dato che, come si è già scritto, *hunz* è un semplice involucro di *gunz*, avviene un'ultima volta quanto avveniva in ogni passaggio del ciclo eterno *while(true)*, **SOLO CHE, QUESTA VOLTA,** il numero di livelli di invocazione di funzioni, ossia la profondità dello *stack* da srotolare, è **TRE** invece di **DUE**.   
  
Quando fosse eseguito *throw* da parte di *funz*, la *throw expression*, lanciata nell'etere maligno, sarebbe catturata dal *catch(...)* che segue il *try* di *hunz*, che è adeguato per natura alla cattura di espressioni di qualunque tipo, pur non potendo farci nulla, non avendo un nome con cui chiamarle. Esso esegue novamente, dal suo interno, l'operatore *throw* **seguito da nulla** (**E TENETE COME CERTO CHE QUESTO MODO DI ESEGUIRE *throw* VALE SOLTANTO ENTRO UN *catch***), il che significa che la *throw expression*, dopo le azioni compiute dal *catch(...)*, viene rilanciata verso nuovi destini, ossia verso un eventuale altro *catch* di livello superiore nello *stack*; il quale, in questo caso, esiste, ed è il *catch(...)* di *main*, che chiude con eleganza il programma.  
  
Tutto quanto fin qui scritto spiega esaurientemente ogni possibile esecuzione del codice proposto.  
  
**Osservazioni aggiuntive:**

* si è sottolineato che le clausole come *throw(int, double)* poste a destra della lista di argomenti di una funzione sono OPZIONALI. In loro assenza è consentita l'esecuzione di *throw* con espressioni di tipo qualunque; se, in presenza di una di tali clausole, venisse eseguito *throw* su un'espressione di tipo non previsto tra quelli inseriti, la *throw expression* coinvolta non sarebbe catturata da alcun *catch*, NEMMENO DA QUELLO coi tre puntini, e il programma si interromperebbe con la funzione di sistema *abort( )* esattamente come accadrebbe, in assenza di clausole *throw(/\*omissis\*/)*, se non fosse fornito alcun *catch* idoneo;
* nell'esempio fornito, se il *catch(...)* che segue *hunz* NON avesse rieseguito *throw*, l'avventura della *throw expression* partita da *funz* si sarebbe conclusa lì, e il *catch* di *main* sarebbe stato saltato: ciò avrebbe fatto apparire il contraddittorio messaggio *"se vedi questo hai digitato un intero positivo!"* anche dopo averne digitato uno negativo (la LOGICA, la LOGICA, ... questa virtù bistrattata...);
* se in *main*, prima del *catch(...)* fossero stati inseriti anche i due *catch* incaricati della cattura di espressioni *int* e *double*, come era stato fatto entro il ciclo, la *throw expression* rilanciata dal *catch(...)* di *hunz* sarebbe stata catturata dal *catch* giusto, e questo nonostante che il *catch(...)* di *hunz* non avesse dato un nome a quell'espressione (**PROVATE PER CREDERE, E MEDITATE A LUNGO SU QUESTO FATTO, PERCHÉ LO RITROVEREMO)**.

**Canto sedicesimo: le funzioni e come si puntano**

Fin dall'inizio di questo percorso si è sempre parlato di funzioni come di parti costitutive di un programma C++ e si è sottolineato già nelle primissime righe dell'introduzione che *main* stessa è una funzione, della quale è impossibile fare a meno e che DEVE avere proprio QUEL NOME e QUEL TIPO, perché è SOLO da quel punto che l'esecuzione di un programma INIZIA.  
  
Anche le funzioni, come le variabili, DEVONO essere dichiarate ma, diversamente dalle variabili, possono anche essere dichiarate più volte nello stesso ambito, anche quando fosse inutile. Per intendersi, una balbuzie logica come la seguente

void funz( );  
void funz( );  
void funz( ) {cout <<"funz\n";}  
  
int main( )  
{void funz( );  
funz( );}

in cui la funzione *funz( )* appare oziosamente dichiarata ben tre volte nell'ambito globale, ma compiutamente definita SOLO una volta, e ridichiarata addirittura per una QUARTA VOLTA nell'ambito di *main*, è candidamente ammessa dal compilatore che invece bollerebbe come errore la ripetizione della dichiarazione di una semplice variabile.  
Inoltre la ridichiarazione di *funz( )* in *main* NON COPRE la dichiarazione nell'ambito globale: quando *main* invoca l'esecuzione di *funz* si tratta proprio di QUELLA *funz* e di nessun'altra, mentre se *main* ridichiarasse una variabile dichiarata nell'ambito globale sarebbe una variabile DEL TUTTO DIVERSA E DISTINTA.  
  
Le ragioni di questa diversità di trattamento, e della conseguente tolleranza mostrata dal compilatore, risiedono nel fatto che NON È LUI a doversi occupare dell'identificazione della funzione che deve essere eseguita quando da qualche parte del programma si richiede l'esecuzione di una certa funzione, bensì il cosiddetto *linker*, ossia il programma residente nel calcolatore che è il diretto responsabile della scrittura materiale del codice eseguibile e che viene mandato in esecuzione automaticamente dal compilatore stesso se e quando non ha riscontrato alcun errore sintattico.  
Dal punto di vista del compilatore, il codice precedente non mostra alcun errore sintattico, perché il linguaggio, di cui il compilatore è fedele interprete, ammette la possibilità della coesistenza di **diverse funzioni con lo stesso nome**, demandando appunto al *linker* la segnalazione di eventuali ambiguità non risolvibili, e che non permettano quindi A LUI di produrre il codice binario eseguibile. E dal momento che, come si è detto, la pluridichiarata (perché pluridichiarabile) *funz* è stata *"compiutamente definita SOLO una volta"* (autocitazione da sopra) NEPPURE il *linker* ha alcun motivo di lagnarsi. Peraltro la ridefinizione, NON la ridichiarazione, della stessa *funz* sarebbe motivo sufficiente AL COMPILATORE per segnalare errore sintattico; non ritenete oro colato questa distinzione tra compilatore e *linker* (utile comunque a scopi pedagogici), perché alla fin fine il programma che genera l'eseguibile viene invocato con un unico comando. Detto con tutto il rispetto dovuto alla profondità del mistero, è, alquanto alla lontana, come se il compilatore fosse simultaneamente uno e trino.  
  
La caratteristica del linguaggio di ammettere che numerose funzioni possano avere lo stesso nome è identificata tecnicamente come *overload* delle funzioni ed è descritta compiutamente in una pagina dedicata.  
  
Tuttavia, per non farvi interrompere il percorso, se ne riprendono anche qui le linee essenziali; potrete poi approfondire a vostro agio i concetti consultando quella pagina.  
  
Non bisogna dimenticare che:

* il compilatore segnala errore SOLTANTO se una funzione dichiarata in maniera IDENTICA viene definita due volte; in altre parole NON LASCIA IMPUNITA una cosa come questa:  
  *void funz( ) {cout <<"ciao\n";}  
  void funz( ) {cout <<"addio\n";}*  
  mentre non batte ciglio, ad esempio, di fronte a una cosa così:  
  *void funz(int i) {cout <<"ciao "<<i<<"\n";}  
  void funz(int &i) {cout <<"addio "<<i<<"\n";}*
* è il *linker* a segnalare errore QUANDO NON SA CAPIRE quale funzione si intenda eseguire; ad esempio, di fronte alle due ultime linee di codice del punto precedente, CHE IL COMPILATORE LASCIA PASSARE, sarà il *linker* a protestare se, da qualche parte (ad esempio in *main*), ci fosse un segmento di codice come questo:  
  *int k = 5;  
  funz(k);*perché non saprebbe se farci dire *ciao* o *addio* seguito da 5.
* Nelle stesse condizioni descritte nel punto precedente, il *linker* si direbbe invece per nulla dubbioso e del tutto appagato di fronte al seguente codice, **APPARENTEMENTE** EQUIVALENTE:  
  *funz(5);*e ci farebbe dire *ciao 5* senza alcun tentennamento.

Accendete tutti i vostri neuroni e pesate ogni sillaba che leggete, perché nel contenuto del precedente elenco puntato sta la **QUINTESSENZA** sia dei meccanismi dell'*overload* delle funzioni sia di come queste ricevono i propri argomenti.  
  
La chiave di ogni comprensione è posta nel segnetto *&* che compare nella dichiarazione della seconda *funz* (quella che direbbe *addio*) e che rende le due dichiarazioni COMUNQUE NON IDENTICHE, per cui è lecito, dal punto di vista del compilatore, che siano definite entrambe.  
Dovreste ricordare (**TUTTO DOVETE RICORDARE!**) che quel segno, se compare in un'espressione, è l'operatore unario che prende, e restituisce a chi lo usa, l'indirizzo in memoria del proprio operando; ma qui NON COMPARE in un'espressione, bensì **NELLA DICHIARAZIONE**, confinata all'ambito della seconda *funz*, dell'argomento che essa riceve. Che cos'è dunque quell'argomento? O, equivalentemente, che cosa si intende quando si usa *i* entro quella funzione? E che cos'è, invece, l'argomento della prima *funz*? O, equivalentemente, che cosa si intende quando si usa *i* entro quest'altra funzione?  
  
Da quando è nato il mondo (o meglio il linguaggio C) scrivere *int i* ha sempre significato che *i* è *una variabile*, del tipo indicato, la cui esistenza è circoscritta all'ambito in cui è posta quella dichiarazione; le parole hanno il loro peso: dire *esistenza circoscritta all'ambito* implica che *i* abbia comunque una PROPRIA esistenza, effimera finché si vuole, che INIZIA appena prima di entrare nella funzione (*i*, in QUELL'ISTANTE, viene **CREATA**) e TERMINA appena prima di uscirne (*i*, in QUELL'ISTANTE, viene **DISTRUTTA**), ma, in ogni caso, PROPRIA, vale a dire indipendente da quella di checchessia altro.  
L'unico contatto di *i* col parametro che viene inserito da *main* (o da qualunque altro chiamante) nelle parentesi tonde al momento dell'invocazione della funzione è il valore che (eventualmente) vi si trova e che viene **ricopiato** in *i* all'atto della sua creazione: da quel momento e fino alla morte *i* vive una vita, forse breve, ma **del tutto autonoma**.  
  
Invece, quando nasce il linguaggio C++, che pure conserva TUTTO quanto è stato appena detto, scrivere *int &i* ha un significato **TOTALMENTE DIFFERENTE**: qui *i* **NON È UNA VARIABILE**, bensì, come è stato detto nel canto sugli operatori, **UN RIFERIMENTO A UNA LOCAZIONE DI MEMORIA CONTENENTE UN *int***.   
Anche qui occorre dar peso alle parole: la frase in grassetto non può significare altro che la memoria cui *i* fa riferimento PER TUTTA LA DURATA DELLA FUNZIONE IN CUI È DICHIARATO **DEVE GIÀ ESISTERE**, altrimenti come si farebbe a farvi riferimento? Per giunta quella locazione di memoria è CONTENENTE UN *int* (autocitazione); e quale può essere questo *int* se non quello trasmesso come parametro quando si invoca la funzione? Per pura deduzione logica si conclude che *i* fa riferimento alla **STESSA** (eventuale) memoria cui ci si riferisce nel chiamante tramite il parametro trasmesso, NON a una memoria diversa CREATA per l'occasione. Pertanto, in questo caso, *i* NON HA AFFATTO UNA VITA PROPRIA, ma CONDIVIDE QUELLA DEL PARAMETRO TRASMESSO e quindi **È IL PARAMETRO TRASMESSO**.  
  
Se avete ben letto e BEN CAPITO i due ultimi capoversi avete capito QUASI TUTTO il linguaggio (quindi è MEGLIO che li rileggiate con più attenzione) e vi dovrebbe apparire LAMPANTEMENTE CHIARO quanto si è detto sopra a proposito dei dubbi e delle certezze del *linker* (ammettendo che possediate una LOGICA grande appena quanto l'evangelico granello di senapa).  
  
In effetti, ferme restando le due dichiarazioni di *funz* in *overload*, quando si invocasse *funz(k);* quali sono le intenzioni del/la programmatore/trice? Eseguire la funzione *funz(int i)*, cui *k* dà come unico contributo il proprio valore che resterà senz'altro intonso al termine della funzione...o eseguire piuttosto la funzione *funz(int &i)*, cui *k* contribuisce sia col proprio valore sia con la propria collocazione in memoria, e che per ciò stesso è **potenzialmente in grado di modificare il valore che *k* possedeva prima che la funzione fosse eseguita**?   
Nessuna delle due scelte possibili è aprioristicamente migliore dell'altra, e quindi il *linker*, non potendo assumersi la responsabilità di operarla, se ne lava le mani segnalando ERRORE.  
  
Invece, quando si scrive *funz(5);*, ogni incertezza non ha più giustificazione e viene eseguita SENZ'ALTRO *funz(int i)* e MAI *funz(int &i)*, per la semplice ragione che *"una costante esplicita entra nel programma direttamente col proprio valore"* (citazione dal canto sulle costanti) e NON HA UNA COLLOCAZIONE PERMANENTE IN MEMORIA cui poter fare riferimento. Pertanto l'unica possibilità residua consiste nel ricopiare il valore della costante nella memoria creata per la nuova variabile *i* che vive in *funz(int i)* (d'altronde, visto che *funz(int &i)* avrebbe la potenzialità di modificare il valore di quello che riceve come argomento, come farebbe a modificare il valore di una costante esplicita?...).  
  
La distinzione fra ricezione del valore o del riferimento alla memoria per quanto riguarda un argomento di una funzione vige anche, in modo del tutto analogo, quando quell'argomento fosse un puntatore, con le conseguenze che, a questo punto, dovrebbe essere facile intuire; si consideri il codice seguente, CHE VI SI RACCOMANDA CALDAMENTE DI FAR ESEGUIRE (non preoccupatevi dei messaggi d'avvertimento che il compilatore vi darà a proposito delle due inizializzazioni del puntatore *p* negli ambiti delle funzioni *funz* e *gunz*; sono dovuti al fatto che esso non può capire che vi siete ampiamente cautelati contro l'eventualità di inizializzazione di memoria non allocata mercé il modo usato per inizializzare l'intero *m*):

# include <iostream>  
using namespace std;  
  
void funz(int \*p, int n)  
{  
cout <<   
"Sono funz: ecco i valori puntati in main\n";  
for(int i=0; i < n; ++i) cout << p[i] << ' ';  
cout << '\n';  
cout << "Ora incremento di uno ciascun valore\n";  
for(int i=0; i < n; ++i) ++\*(p+i);  
cout <<   
"Adesso riinizializzo il puntatore come mi pare\n";  
int m = n < 4 ? 4 : n;  
p = new int[m] {10, 20, 30, 40};  
cout << "Ecco che cosa punta adesso\n";  
for(int i=0; i < m; ++i) cout << p[i] << ' ';  
cout << '\n';}  
  
void gunz(int \*&p, int n)  
{  
cout <<   
"Sono gunz: ecco i valori puntati in main\n";  
for(int i=0; i < n; ++i) cout << p[i] << ' ';  
cout << '\n';  
cout << "Ora incremento di uno ciascun valore\n";  
for(int i=0; i < n; ++i) ++\*(p+i);  
cout <<   
"Adesso riinizializzo il puntatore come mi pare\n";  
int m = n < 4 ? 4 : n;  
p = new int[m] {10, 20, 30, 40};  
cout << "Ecco che cosa punta adesso\n";  
for(int i=0; i < m; ++i) cout << p[i] << ' ';  
cout << '\n';}  
  
int main( )  
{  
int a[ ] {1, 2, 3, 4, 5};  
int \*b = new int[3] {100, 200, 300};  
funz(a, 5);  
gunz(b, 3);  
cout <<   
"Sono main. Tutte le funzioni sono state eseguite.\n"  
<< "Ecco il contenuto dell'array a:\n";  
for(int i=0; i < 5; ++i) cout << a[i] << ' ';  
cout << "\ned ecco che cosa punta il puntatore b:\n";  
for(int i=0; i < 3; ++i) cout << b[i] << ' ';  
cout <<   
"\n(non è che magari ce ne sia anche un altro ?)\n";  
cout << b[3] << "\nehilà, c'era davvero...\n";  
}

Le due funzioni *funz* e *gunz* differiscono SOLO per la maniera di ricevere il loro primo argomento (e per come si presentano al mondo, ma questa è una pignoleria utile soltanto a riconoscerle); per il resto SONO TOTALMENTE IDENTICHE, come ci si può persuadere facilmente leggendole con attenzione. Eppure quell'unica differenza ha conseguenze formidabili che risultano chiare se si esamina il comportamento del codice una volta eseguito.   
Quando *main* invoca l'esecuzione di *funz* le trasmette il nome dell'*array* (ossia *a*) e la sua capienza, e di questi due parametri *funz* riceve **SOLO** il valore, perché nella lista dei suoi argomenti NON appare mai il segno *&* (per inciso si vede anche che una funzione può TRANQUILLAMENTE ricevere come puntatore ciò che le viene trasmesso come *array*, a conferma di cose già note).  
Ora però si sa (o si dovrebbe sapere) che il valore di un puntatore è l'indirizzo della memoria in cui risiedono le variabili che esso punta e quindi, per quanto è stato detto sopra, non appena *funz* inizia a eseguirsi, in essa VIVE DI VITA PROPRIA un puntatore *p* **completamente diverso e distinto** da *a* (che vive in *main*), ma che ha **IL SUO STESSO VALORE** (almeno nel momento in cui la funzione si avvia) e quindi **STA PUNTANDO LE STESSE VARIABILI** (quelle che si trovano pure loro in *main*).   
Per le ragioni APPENA DETTE, quando *funz* incrementa di 1 le variabili puntate dal SUO puntatore sono le variabili contenute in *a* a incrementarsi (e infatti *main* se le ritrova tutte incrementate), mentre quando reinizializza il SUO puntatore ad *a* NON SUCCEDE UN BEL NULLA, non essendo LUI a essere reinizializzato (e del resto neppure potrebbe, essendo un *array*, RICORDATE?).  
  
Viceversa *gunz* riceve il proprio primo argomento come riferimento in memoria (per inciso, QUELLA è la posizione in cui scrivere il segno *&* quando l'argomento è un puntatore, perché si tratta di una DICHIARAZIONE, non di un'espressione, e in una dichiarazione i segni *\** e *&* NON sono interscambiabili, dato che non avrebbe senso dichiarare un puntatore a *int&*...) e quindi, per quanto si è ampiamente dibattuto, riceve il parametro trasmesso *in corpore vivo* e non solo per ricopiatura di valore. Ne consegue che l'incremento di 1 dei contenuti va completamente perduto, assieme ai valori iniziali stessi così come impostati da *main*, nel momento preciso in cui viene eseguita la reinizializzazione del puntatore che, pur avvenendo entro *gunz*, tocca PERSONALMENTE il puntatore *b* dichiarato in *main*: in effetti colà appaiono i valori inizializzanti dati da *gunz*, NON PIÙ AFFATTO quelli che c'erano prima dell'esecuzione della funzione; anche il numero di variabili puntate è coerentemente cambiato, come recita con sorpresa (relativa) *main* stessa nei suoi ultimi messaggi.  
  
Parrebbe inutile dover sottolineare che in questo esempio si sono usati nomi diversi per le funzioni (*funz* e *gunz*) giustappunto per non cadere nello stesso errore già discusso in questo stesso canto allorché si trasmettevano alle funzioni semplici variabili; altresì appare ovvio dire che funzioni omonime la cui lista di argomenti differisca per numero, o per tipo, o per entrambe le cose, possono sempre coesistere nello stesso programma senza alcuna difficoltà; altrettanto dovrebbe essere evidente che il tipo restituito dalle funzioni non può bastare a dirimere le ambiguità in cui il *linker* può incorrere: in altre parole queste due semplicissime funzioni omonime  
  
*int funz( ){return 1;}  
float funz( ){return 1.f;}*genereranno SEMPRE errore a ogni loro invocazione nello STESSO programma.  
  
Qualche parola in più meritano le funzioni dotate di argomenti cosiddetti *standard*.  
  
Si dice *standard* un argomento di una funzione che le può essere trasmesso, come qualsiasi altro, ... ma anche no, dato che, qualora non fosse trasmesso, verrà inizializzato nell'ambito della funzione con un valore predeterminato dal programmatore.  
Ogni funzione può avere simultaneamente quanti si vogliano argomenti NON *standard* e quanti si vogliano argomenti *standard*, ma i secondi, QUANDO CI SONO, DEVONO ESSERE **TUTTI** POSPOSTI a **TUTTI** i primi nella lista complessiva degli argomenti.   
Un argomento si riconosce essere *standard* se è inizializzato col valore predisposto dal programmatore DIRETTAMENTE NELLA LISTA DEGLI ARGOMENTI. Ad esempio:  
  
*void funz(int i = 2);*è la DICHIARAZIONE di una funzione di tipo *void* con un solo argomento che è un argomento *standard*. Quando, come nell'esempio dato, la dichiarazione della funzione è SEPARATA dalla sua definizione, la specifica di quali siano gli (eventuali) argomenti *standard* va fatta UNA SOLA VOLTA, O nella dichiarazione O nella definizione, **NON** in tutte e due. Ciò significa che la seguente scrittura è un ERRORE  
  
*void funz(int i = 2);  
void funz(int i = 2) {  }*mentre è corretta CIASCUNA delle quattro seguenti:

1. *void funz(int i = 2);  
   void funz(int i) {  }*
2. *void funz(int = 2);  
   void funz(int i) {  }*
3. *void funz(int i);  
   void funz(int i = 2) {  }*
4. *void funz(int);  
   void funz(int i = 2) {  }*

con quelle scritte ai posti di numero pari da preferirsi a quelle scritte ai posti di numero dispari, per l'inutilità della dichiarazione del nome *i* nelle dichiarazioni della funzione avulse dalla definizione.  
Naturalmente il problema non si pone quando dichiarazione e definizione sono contestuali, come in  
  
*void funz(int i = 2) {  }*   
  
senza nessuna previa dichiarazione. La *funz* di cui si è appena parlato può essere eseguita sia invocandola in modo normale, ossia fornendole il parametro intero che si aspetta, come in *funz(55);* (e in tal caso l'inizializzazione *standard* dell'argomento è semplicemente ignorata) sia omettendo di trasmetterle il/i parametro/i corrispondente/i al/agli argomenti *standard*, come in *funz( );* (e in tal caso la funzione attribuirà al proprio argomento, NEL PROPRIO AMBITO, il valore *standard* inizializzante).  
  
A questo punto dovrebbe apparire chiaro che, dal punto di vista dell'*overload*, una *funz* cotale non può coesistere, nello stesso programma, né con una funzione omonima che riceva un solo argomento non *standard* di tipo *int* (e questo è ovvio, in virtù di quanto scritto nell'elenco puntato) e neppure con una *funz* che abbia la lista di argomenti vuota, perché, DI FATTO, ne ha già occupato LEI la nicchia ecologica. In pratica, quando si deve decidere quali funzioni omonime possano coesistere in *overload*, si deve sempre procedere AL NETTO degli eventuali argomenti *standard*, vale a dire che   
  
*void funz(int i, double k = 3.14, char c = 'a') {  }*occupa, in un sol colpo, i posti competenti a:  
  
*...... funz(int, double, char);  
...... funz(int, double);  
...... funz(int);*  
che pertanto sono, TUTTE E TRE, escluse dalla possibilità di concorrere all'*overload* per la semplice ragione che la funzione provvista di argomenti *standard* potrebbe essere invocata in CIASCUNO dei tre modi in cui sarebbero invocate ANCHE le tre ultime funzioni (al posto dei puntini DOVREBBE andare un tipo QUALSIASI, tanto, come si è detto, il tipo di una funzione NON CONCORRE MAI all'*overload*).   
  
Giova ripetervi di non commettere un **ORRORE** come questo?  
  
*void funz(int i = 1, double k) {  }*   
  
Una funzione può essere trasmessa come argomento a un'altra funzione, realizzando in tal modo una vera e propria *funzione di funzione*, e un simile argomento può anche essere *standard*. Inoltre una funzione può anche essere puntata da un puntatore adeguatamente dichiarato, o riferita attraverso un riferimento. Si discuteranno ora dettagliatamente queste ultime affermazioni.  
  
Per capire il concetto si potrebbe partire dicendo che il nome di una funzione è sostanzialmente esso stesso un puntatore che la punta, o, meglio ancora, un puntatore costante inizializzato nell'atto di definizione della funzione, ed è per questo, in ultima istanza, che una funzione può essere definita UNA VOLTA soltanto. Considerate il codice seguente:

# include <iostream>  
# include <cmath>  
using namespace std;  
  
void funz(double s(double) = sin)  
{cout << s(M\_PI) << '\n';}  
  
int main(  )  
{  
funz(  ), funz(cos);  
}

La funzione *funz* è dichiarata (e definita) come ricevente un unico argomento *standard* che appare a sua volta dichiarato come una funzione ricevente un solo argomento di tipo *double*, e come tale è utilizzato nell'unica espressione che *funz* contiene (*M\_PI* è una costante definita [RICORDATE?] nel documento *cmath* incluso e ha il valore di pi greco). Il valore *standard* dell'argomento è il nome della funzione *sin*, dichiarata e definita in *cmath*, ossia la funzione trigonometrica *seno* (si badi bene: SOLO il nome, NON il nome seguito dalla coppia di parentesi entro cui, comunque, non si avrebbe nulla da poter inserire).  
Questo dovrebbe bastare a spiegare come mai, quando *main* invoca due volte di seguito l'esecuzione di *funz*, come indicato, appaiano sul terminale i valori del seno e del coseno di pi greco (se eseguite il codice [**ESEGUITELO!**] NON spaventatevi nel vedere che il valore del seno non è zero: quello che ottenete è il cosiddetto zero numerico che non coincide quasi mai con quello analitico).  
  
Il precedente programma, anche se istruttivo, non è poi così utile; appena un po' più significativa è la seguente variante:

# include <iostream>  
# include <cmath>  
using namespace std;  
  
double funz(double s(double), double x)  
{return s(x);}  
  
int main(  )  
{  
cout   
<< "verifica dell'identità trigonometrica "  
<< "1/cos^2 = 1 + tan^2 \n";  
for(int i=0; i <= 20; ++i)  
{  
double z = 0.499\*M\_PI\*(i/10.-1);  
cout <<"per x = " <<z  
<<" si ha "   
<< 1.0/(funz(cos, z)\*funz(cos, z))   
<< " da confrontare con "  
<< 1.0 + funz(tan, z)\*funz(tan, z) <<'\n';  
}  
}

**ESEGUITE E MEDITATE** (specialmente su come si fanno i cicli per esplorare un intervallo reale, sul fatto che NON bisognerebbe toccare +/- pi greco/2 [quantunque non sarebbe successo nulla; provate e vedrete] e sul fatto che il nome delle variabili NON importa: si è fatto apposta a dichiarare *z* invece di *x*).  
  
Fin qui si è usato direttamente il nome di una funzione per trasferirlo come argomento a un'altra, e si è detto che si tratta efficacemente di un puntatore costante alla funzione stessa. Costante significa che reinizializzarlo è impossibile, perché sarebbe un errore (ma del resto CHI MAI potrebbe ritenere UTILE a qualcosa una reinizializzazione come *sin = cos;* ?? In ogni caso, se ci provaste, il compilatore vi sputerebbe in faccia...).  
Per la verità pignolesca la nostra *funz* potrebbe lecitamente riassegnare, nel proprio ambito, l'argomento *s* ricevuto scrivendo, per esempio, *s = sin;*, ma ALLORA quale sarebbe l'utilità di *funz*? [DOMANDA A BRUCIAPELO: **perché *funz* lo potrebbe fare?** Risposta in fondo alla pagina, ma provate a rispondere da soli.].  
Noi però sappiamo, oramai, che la vera forza dei puntatori consiste nel poter essere riassegnati ogni volta che si vuole; ed è per questo che il linguaggio fornisce al programmatore la possibilità di dichiarare dei SERI puntatori (intendendo con questo NON costanti) che siano capaci di puntare funzioni.  
Una sintassi tipica per la dichiarazione di un puntatore a funzione è la seguente:

void (\*punt) ( );

Si osservi la posizione delle parentesi ATTORNO a *\*punt* che DEVONO esserci ed essere messe lì per evitare che il compilatore fraintenda e pensi piuttosto che si sta dichiarando una VERA funzione che restituisce un puntatore a *void* (tale sarebbe, in effetti, una dichiarazione come *void \* punt( );*). Si sgombri anche subito il campo da potenziali equivoci: la precedente dichiarazione dice che *punt* è un puntatore capace di puntare TUTTE e **SOLE** le funzioni di tipo *void* e SENZA argomenti; se si vogliono puntare funzioni con una diversa segnatura OCCORRE CAMBIARE COERENTEMENTE la dichiarazione del puntatore. In altre parole NON ESISTE UNA DICHIARAZIONE UNIVERSALE DEI DIRITTI DEL PUNTATORE, in base alla quale gli si possa consentire di puntare QUALSIASI funzione.  
Ciò detto, *punt* potrà essere inizializzato in qualsiasi momento e quante volte si voglia, beninteso SOLO negli ambiti interni a quello in cui è stato dichiarato, con il nome di qualsiasi funzione che corrisponda alla sua segnatura (o prototipo che dir si voglia). Ecco un codice esempio:

# include <iostream>  
# include <cmath>  
using namespace std;  
  
double funz(double (\*)(double), double);  
double (\*punt) (double) = nullptr;  
double (\*qunt)(double (\*)(double), double) = funz;  
  
double funz(double s(double), double x)  
{return (\*s)(x);}  
double gunz(double (\*s)(double), double x)  
{cout << s(x) << '\n';  
return 0.0;}  
  
int main(  )  
{char c;  
double x;  
cout << "quale funzione vuoi ? s = seno, c = coseno\n";  
while(!punt) {  
cin >> c;  
switch(c)  
{case 's': punt = sin; break;  
case 'c': punt = cos; break;  
default: cout << "scelta non valida; rifare\n";}}  
cout << "digita un numero reale ", cin >> x;  
if(x <= 0.0)  
{qunt = gunz, cout << "da gunz ";  
(\*qunt)(punt, x); return 0;}  
cout << "da funz "<< qunt(\*punt, x) << '\n';   
}

Ecco che cosa si impara dal programma precedente:

1. un puntatore a funzione è l'unico puntatore che coincide con la propria dereferenza, eccetto, ovviamente, quando lo si inizializza; in effetti il programma mostra che quando viene usato per invocare l'esecuzione della funzione che punta, o anche quando è inserito nella lista degli argomenti di una funzione, è indifferente scrivere *punt* o *(\*punt)* se *punt* è il nome del puntatore;
2. il puntatore *qunt* è senz'altro inizializzato DUE volte quando si digita un numero negativo, a conferma che non si tratta di un puntatore costante; inoltre, la prima volta, è inizializzato con *funz* PRIMA che questa sia definita. Ciò è perfettamente legittimo, perché *funz* è già un indirizzo nel momento in cui è dichiarata. Basta solo che sia definita nel momento in cui il puntatore pretende di eseguirla, come fa nell'ultima riga del codice;
3. *nullptr* (parola del vocabolario) è il puntatore nullo per antonomasia e il suo valore può essere assegnato a qualsiasi puntatore (ma proprio QUALSIASI); il programma se ne giova per rifiutare sdegnosamente la digitazione di qualsiasi carattere diverso tanto da 's' quanto da 'c';
4. quando si dichiara una funzione che ha tra i propri argomenti un puntatore a funzione, e non la si definisce (come avviene con la nostra *funz* alla prima riga di codice), l'omissione dei nomi degli argomenti si traduce, per quanto riguarda il puntatore a funzione, nella comparsa dell'inconsueto costrutto di un asterisco racchiuso tra parentesi; ciò è del tutto coerente con la sintassi (peraltro il costrutto *(\*)* potrebbe anche essere omesso del tutto);
5. si sottolinea che, in *main*, né il nome *funz* né il nome *gunz* sono mai usati per eseguire le due funzioni.

Tutto il resto sono cose che dovreste già sapere. Concludiamo con alcune puntualizzazioni, alcune delle quali sono dovute a puro amor di pignoleria.

* quando si dichiara un puntatore a funzione è perfino ammissibile, anche se del tutto inutile, aggiungere un ambito vuoto, MA IN TAL CASO NON SI PUÒ COMPIERE L'IMMEDIATA INIZIALIZZAZIONE. Pertanto, pur non essendo un errore, è caldamente sconsigliato visto che apporta potenziali danni senza alcun beneficio.  
  Ad esempio è lecito scrivere  
  *double (\*punt) (double){  };*ma NON   
  *double (\*punt) (double){  } = nullptr;*
* è lecito dichiarare *array* di puntatori a funzione, i cui elementi potranno essere inizializzati in modo del tutto indipendente uno dall'altro, come in:  
  *int main(  ){  
  double (\*punt[2]) (double) {sin, cos};  
  // omissis  
  }*
* e perfino puntatori a puntatori a funzione (senza limiti al numero di asterischi), che potranno essere inizializzati a piacimento, come in  
  *int main(  ){int n;  
  double (\*\*qunt) (double);  
  cout << "di quanti puntatori hai bisogno ? ", cin >> n;  
  qunt = new (double(\*[n])(double));  
  // inizializzare da qunt[0] a qunt[n-1]  
  // magari con uno switch discendente e senza break  
  // PROVATECI  
  }*da notare che, in un caso simile, nell'inizializzazione di *qunt*, la coppia di parentesi tonde il cui esemplare destro sta immediatamente prima del punto e virgola È OBBLIGATORIA.
* Se una funzione ha argomenti *standard*, per poter essere puntata da puntatori sfruttandone le caratteristiche, ce ne vorrebbe uno per ogni diversa possibile invocazione della funzione (e quindi si perde l'utilità O dei puntatori O degli argomenti *standard*...); in altre parole NON È POSSIBILE inserire argomenti *standard* nella dichiarazione di un puntatore a funzione.

Come approfondimento e/o conferma di quanto detto in questo canto si suggerisce la lettura, una volta completato il percorso, del documento sulla *function signature* reperibile dalla pagina regole generali del C++.

**Risposta alla DOMANDA A BRUCIAPELO:**  
*perché nell'ambito della funzione l'argomento non è il puntatore originale, ma una sua copia autonoma che morrà con la funzione stessa; non ricordate la regola di ricezione degli argomenti da parte di una funzione? Diverso sarebbe stato se la funzione fosse stata definita così:****double funz(double (&s)(double), double x)  
{return s(x);}***ove ancora si sottolineano le due parentesi ATTORNO a **&s**, che ne fanno un riferimento sinistro a funzione, quale si era anticipato in questo stesso canto che potesse esistere.

**Terza pausa di riflessione**

Dopo aver riletto con attenzione tutti i canti fin qui proposti (dal canto 0 al canto 16), SE LI AVETE TUTTI BEN COMPRESI dovreste aver completato la conoscenza del linguaggio dal punto di vista PROCEDURALE, mancando solo una trattazione più approfondita del problema dell'INPUT/OUTPUT che sarà oggetto dei prossimi canti.  
Cimentatevi nella scrittura di un VOSTRO programma che contempli TUTTO (o quasi tutto) quello che dovreste aver imparato finora, che serva a risolvere un problema semplice di Fisica o di Matematica e che sia strutturato in funzioni.  
Ricordatevi che siete aspiranti Fisici e che, pertanto, problemi da affrontare ne dovreste conoscere a iosa.   
  
**FATELO SUL SERIO, SE VOLETE IMPARARE**, altrimenti sarà peggio per voi.   
  
Quando l'avrete fatto, e SOLO quando vi funzionerà, potrete andare avanti nella lettura: il bello deve ancora venire.

**Canto diciassettesimo: il problema dell'INPUT**

Si tratta del problema di come fa un programma ad acquisire dati dall'esterno e fino a questo punto è stato affrontato in maniera molto, ma mooooooolto, marginale, giusto perché non poteva essere ignorato del tutto.  
In effetti ci siamo sempre limitati a leggere dati da tastiera, con l'ausilio dell'oggetto *cin* e dell'operatore *>>* (che, in tale contesto, era stato denominato estrattore), in espressioni del tipo  
  
*cin >> n;*   
  
che sono state introdotte fin dagli esordi di questo percorso di conoscenza, ma è del tutto EVIDENTE che questa procedura non può essere sufficiente, essendo del tutto inadeguata alla manipolazione di quantità di dati che possono raggiungere i Mega, i Giga, e perfino i Tera *bytes* (vale a dire l'equivalente di mille miliardi di tasti premuti sulla tastiera: c'è qualcuno che se la sente?).   
  
Accanto al problema dell'INPUT esiste anche quello dell'OUTPUT, ossia di come far sì che un programma comunichi i propri risultati a chi lo usa: si tratta di un problema ESTREMAMENTE più semplice che tratteremo successivamente, e che finora era stato brillantemente risolto tramite l'oggetto *cout* e l'operatore speculare *<<*, detto inseritore, di cui sono ZEPPI tutti i codici visti prima di adesso (e che soffrono della stessa inadeguatezza).  
  
Qualunque sia il verso di percorrenza dei dati (INPUT: dall'esterno al programma; OUTPUT: dal programma all'esterno), per il linguaggio si tratta SEMPRE di una sequenza, o flusso ordinato che dir si voglia, di *bytes*: si parla infatti, tecnicamente, di *input stream* (o di *output stream*); non per nulla il nome del documento che DA SEMPRE si include all'inizio dei codici recita *iostream*.  
  
**LA PRIMA COSA DA METTERSI IN TESTA È QUINDI CHE IL SUPPORTO FISICO DEI DATI NON INFLUENZA LA LORO VISIONE DA PARTE DEL PROGRAMMA**, se non per il fatto che esistono supporti su cui, per loro natura FISICA, non è possibile tornare indietro (uno di questi è proprio la tastiera: una volta che si è premuto un tasto non si può fingere di non averlo premuto, un po' come diceva Agnese ne "I Promessi Sposi" a proposito di *"lasciar andare un pugno a un cristiano"*).  
  
Limitandosi agli *input stream*, in base al titolo del canto, i loro supporti fisici più frequenti sono:

1. la tastiera (per piccole quantità di dati e per la gestione interattiva dell'esecuzione);
2. il disco fisso (o i dischi fissi) del calcolatore, e segnatamente un (o più di un) particolare documento ivi contenuto (per grandi quantità di dati);
3. ogni genere di supporti rimovibili: CD, DVD, Bluray, chiavette/dischi USB e via dicendo (per cospicue moli di dati, magari provenienti da ALTRI calcolatori);
4. la memoria del calcolatore (per lettura di dati ivi residenti, e magari appartenenti allo stesso programma in esecuzione);
5. la scheda video del calcolatore (per lettura di immagini o filmati dallo schermo e per la gestione interattiva dell'esecuzione tramite *mouse* o altri dispositivi di selezione/puntamento);
6. la scheda audio del calcolatore (per lettura di dati musicali e/o parlati);
7. la scheda di rete del calcolatore (per lettura di dati provenienti da internet);
8. varie ed eventuali (ad esempio l'orologio di macchina, che, pur non potendo definirsi in senso stretto un supporto di *input stream*, consente comunque al programma di ottenere informazioni di un certo pregio quali l'ora del giorno o la data del mese; oppure il cosiddetto *environment*, da cui il programma può ricavare informazioni sulla macchina che lo esegue o sulla persona dell'utilizzatore).

Come si vede ce n'è per tutti i gusti e le esigenze, ma si sottolinea ancora una volta che il linguaggio tratta allo STESSO modo TUTTE le voci dell'elenco: una *fila* unidimensionale e ininterrotta di *bytes* con un inizio (un *primo byte*) e una fine (un *ultimo byte*). Probabilmente si sta dando per scontato che sappiate che cosa sia un *byte*: se avete dei dubbi domandatelo a lezione al vostro docente.  
  
Dove stanno le differenze, per cui è stato possibile scrivere un elenco composto di numerosi punti? Si è già accennato al fatto, ad esempio, che alcune di queste *file*, o *input stream*, possono essere scansionate in un solo verso: DALL'inizio ALLA fine; altre invece consentono il riavvolgimento.  
Alcuni di questi *stream* possono essere scansionati solo in maniera sequenziale, ossia non è possibile raggiungere un certo *byte* della fila senza attraversare tutti quelli che lo precedono nella sequenza (tipicamente, ancora una volta, la tastiera, ma anche la scheda audio o quella di rete); altri invece lo permettono e sono allora detti *ad accesso diretto* (tipicamente i dischi o la memoria).  
Da ultimo, in alcuni *stream* l'inizio e la fine hanno una collocazione ben precisa e individuata (un documento su disco, per esempio), mentre per altri questi concetti non sono altrettanto precisamente individuabili e sono collocati solo in seguito a un atto di volontà (la tastiera, per esempio) o a un puro accidente (l'interruzione o il ripristino di un collegamento di rete, ad esempio).  
  
Tutti gli *input stream*, su qualunque supporto fisico risiedano, per poter essere utilizzati devono, per prima cosa, ESSERE APERTI: questa frase significa semplicemente che il programma deve, per prima cosa, prendere coscienza della loro esistenza; il modo di effettuare l'apertura di un *input stream* è un'altra delle differenze che distinguono gli *stream* uno dall'altro secono il supporto in cui risiedono, anzi, si può dire che, dal punto di vista del programmatore, questa è la differenza più significativa.  
  
Una volta APERTO, l'*input stream* è disponibile a fornire al programma i dati che contiene sotto forma, giova ripeterlo, di una sequenza ordinata di *bytes*; il programma, a sua volta, mantiene traccia in una propria variabile interna (di valore accessibile, su richiesta, al programmatore e che chiameremo indice di lettura o semplicemente indice quando non vi sia ambiguità) della posizione raggiunta lungo lo *stream*, vale a dire di quale *byte* della sequenza è passibile della prossima operazione di presa dati.  
  
**OGNI PRESA DATI (LETTURA) CONCLUSA CON SUCCESSO AGGIORNA AUTOMATICAMENTE IL VALORE DELL'INDICE DI LETTURA**, il che significa

1. che non deve pensarci il programmatore, ovvero che, quando una lettura riesce, lo *stream* è IMMEDIATAMENTE PRONTO per una lettura successiva, nel verso inizio ---> fine;
2. che il numero di *bytes* di cui l'indice avanza lungo lo *stream* è esattamente uguale al numero di *bytes* trasferiti dallo *stream* al programma e che QUEL numero di *bytes* non fa oltrepassare la fine dello *stream*, altrimenti la lettura non si sarebbe potuta concludere con successo;
3. che uno *stream* APPENA APERTO CON SUCCESSO ha l'indice di lettura posizionato sul PRIMO *byte* (e vale quindi 0 (zero)) [o su quello che è da considerare il primo *byte* subito dopo la riuscita apertura, come potrebbe accadere nel caso di uno *stream* supportato da una connessione di rete].

Viceversa ogni LETTURA FALLITA **LASCIA INVARIATO L'INDICE DI LETTURA** e simultaneamente viene attivata una sorta di sistema di allarme (sirene, campanelli, triccheballacche...) di cui l'avveduto programmatore può giovarsi per prendere le opportune contromisure.  
Si sottolinea qui che, quando il sistema di allarme di un *input stream* è, per qualsiasi ragione, nello stato ATTIVATO, l'*input stream* stesso sospende la propria operatività per cui ogni ulteriore tentativo di lettura compiuto **SENZA AVER PRIMA DISATTIVATO** il sistema di allarme è destinato a fallire a sua volta, generando evidentemente una spirale viziosa da cui NON SI ESCE. Quanto appena detto porta alle seguenti, ovvie, conclusioni:

1. gli errori **NON SI FANNO**. In questo modo il sistema di allarme MAI si attiva e tutto fila liscio; equivale a supporre, ad esempio, che davanti alla tastiera si sieda SEMPRE un *homo sapiens sapiens* in stato di perfetta sobrietà e lucida concentrazione e MAI un *papio papio* (babbuino della guinea) imbottito di cocaina;
2. nella remota ipotesi che il predetto *homo sapiens sapiens* incorra in una deprecabile *défaillance*, egli/ella stesso/a, recuperata rapidamente la pienezza delle proprie facoltà intellettive, DOVRÀ adoprarsi per porre rimedio al proprio misfatto, e dovrà AVERLO FATTO PRIMA di esservi incorso/a.

Lo stravolgimento del principio di causa/effetto che sembra adombrato nell'ultima affermazione è solo apparente quando si parla di scrittura di programmi: in realtà significa che il programma stesso dovrà essere stato scritto (quindi PRIMA di usarlo) tenendo presenti sia i propri limiti sia l'ineluttabilità della legge di Murphy.  
  
Per rendersi persuasi del rischio di spirale viziosa cui si è fatto cenno sopra, basterà eseguire il seguente, elementarissimo, in apparenza innocentissimo e certamente inutilissimo, programma eterno:

# include <iostream>  
using namespace std;  
  
int main(  ) {  
int n;  
while(true) {  
cout << "dimmi un numero intero\n";  
cin >> n;  
cout << "mi hai dato " << n << '\n';  
}}

e supporre che l'*homo sapiens sapiens* alla tastiera si alzi per andare a bere un caffè e sia sostituito, di soppiatto, dal *papio papio*.   
Comportatevi dunque, volontariamente, da *papio papio* (solo per un fuggevole istante, e SENZA COCAINA, mi raccomando) e vedrete che cosa vi succederà. Naturalmente l'esempio è solo didascalico perché comunque, per terminare il programma, anche l'*homo sapiens sapiens* non avrebbe potuto far altro che inviare il segnale d'interruzione CTRL-C.  
  
Dopo tutte queste considerazioni introduttive e generali, che comunque vi si raccomanda di tenere sempre ben presenti, entriamo *in medias res*.  
Nel linguaggio sono contemplati almeno tre sistemi completi per la gestione del problema dell'INPUT (e dell'OUTPUT): il sistema Unix/Posix, forse uno dei più antichi, tuttora presente, ma ormai quasi disusato e fruibile includendo il documento *unistd.h* (si basa su funzioni e sui cosiddetti *file descriptors*; non ce ne occuperemo); il sistema stdio, in uso ai tempi del C, e tuttora fruibile includendo il documento *cstdio* (si basa su funzioni e sulla struttura *FILE*; è ampiamente descritto in pagine dedicate di questo sito); e, buon ultimo, il sistema ios, nato col C++, fruibile includendo almeno il documento *iostream*, come si è SEMPRE fatto, e, in subordine, i documenti *fstream*, *sstream* e *iomanip*, come impareremo a fare (si basa su **OGGETTI** e sarà quello di cui tratteremo diffusamente in questo percorso).  
  
In linea di principio un programma può usare TUTTI i sistemi citati, con solo minime cautele atte a risolvere facilmente eventuali piccoli problemi di sincronizzazione in caso di concorso sullo stesso dispositivo fisico di *stream* gestiti da sistemi diversi; ciò garantisce la piena compatibilità con codici antichi e storicamente consolidati. Tuttavia i codici che vengono sviluppati oggi *ex novo* dovrebbero tendere a privilegiare il sistema *ios*, anche se non mancano nostalgici degli altri sistemi che, anche per questo, sono mantenuti.  
  
Urge soffermarsi sulla sottolineatura e sul corpo grassetto della parola OGGETTI; sta per giungere il momento di parlarne più diffusamente, dato che avrete notato che costituiscono, al posto delle funzioni, il fondamento del sistema *ios*; stiamo per fare il passo dalla programmazione procedurale (quella basata sulle funzioni) alla programmazione cosiddetta *object oriented* (vuol dire appunto basata sugli oggetti). Si entrerà nell'agone tra due canti, dopo aver riservato il prossimo a una breve discussione generale sul problema dell'OUTPUT, tanto per pareggiare il canto presente.

**Canto diciottesimo: il problema dell'OUTPUT**

Molto di quanto scritto nel canto precedente vige anche in questo: la definizione di *stream* persiste intatta, il concetto di indice di lettura è sostituito con quello, perfettamente equipollente, di indice di scrittura, l'elenco dei possibili supporti fisici idonei a un *output stream* è identico a quello dato per gli *input stream* tranne l'ovvia sostituzione della tastiera con la finestra della *shell* da cui il programma viene eseguito, il fatto che se si volesse trattare la scheda di rete come *output stream* occorrerebbe che il calcolatore che esegue il programma fosse configurato come *web server* autorizzato e quello che i dispositivi rimovibili utilizzabili come *output stream* non possono essere, tanto per dirne una, dei CD finalizzati...  
  
Anche un *output stream* deve essere, per PRIMA COSA, APERTO, per poter essere utilizzato, ma qui appaiono le differenze sostanziali rispetto a un *input stream*:

1. un *output stream* ha bensì un inizio, ma MAI una fine prefissata, che invece hanno SEMPRE, ad esempio, gli *input stream* supportati dai documenti situati sul disco fisso. Ciò consente di scrivere dati su un *output stream* praticamente all'infinito, senza limitazioni logiche, ma solo limitazioni fisiche, dovute, per esempio, alla capienza del disco. La fine dell'*output stream* è decretata, in ultima analisi, solo dal fatto che si smette di scriverci dentro e si CHIUDE lo *stream*.
2. le affermazioni del punto precedente hanno conseguenze formidabili e potenzialmente pericolose per lo *scriptor inops* (nuova specie animale i cui fossili più antichi non sono ancora stati reperiti, e comunemente chiamata *programmatore sprovveduto*). In effetti l'apertura di un *output stream* implica il posizionamento dell'indice di scrittura all'inizio, ossia nella posizione in cui dovrà andare il PRIMO *byte* che sarà scritto; la fine NON c'è, semplicemente perché un *output stream* APPENA APERTO è, ONTOLOGICAMENTE, **VUOTO** (in questo caso la coniugazione al tempo futuro del verbo scrivere è sostanziale, non solo volitiva come per il verbo leggere relativo a un *input stream*). Ma se fosse reso supporto fisico di quello *stream* VUOTO un documento PREESISTENTE nel disco fisso...ahi ahi ahi... significa che TUTTO IL CONTENUTO DI QUEL DOCUMENTO finisce per andar a fare una passeggiata sui viali...
3. Per le stesse ragioni esposte nel punto precedente aprire un *output stream* rendendone supporto fisico un documento INESISTENTE NON COMPORTA ALCUN ERRORE (come invece accadrebbe se se ne facesse supporto di un *input stream*); tutto è sempre governato da una logica ferrea: quel documento INESISTENTE inizierà semplicemente a esistere sul disco (e VUOTO, come è logicamente giusto che sia, fino a quando non si scriverà qualcosa nello *stream*.)
4. In definitiva, scrivere è molto diverso da leggere...

Qualcuno, a questo punto, potrebbe, con qualche giusta ragione, obiettare che non pare possibile scrivere dati IN AGGIUNTA a quelli già contenuti in un certo documento su disco...  
In realtà non è così: è solo che, quando ci si trovasse davanti a tale esigenza (perfettamente legittima, sia chiaro) NON BISOGNA FARE di quel documento il supporto di un *output stream* puro e semplice, ma di uno *stream* di natura leggermente diversa, come vedremo fra qualche canto.  
  
Una volta messo in testa il contenuto del canto presente il problema dell'OUTPUT non sarà più tale, riducendosi in definitiva alla sola ricerca di un modo per ottenere in uscita delle liste di numeri ben incolonnati (cosa peraltro del tutto secondaria, e di cui, sinceramente, l'autore di questo percorso nemmeno avverte l'urgenza, per programmi che siano di natura scientifica e non ragionieristica).   
  
Che questo non avvenga spontaneamente è insito nel concetto di *stream*: infatti, se eseguirete questo codice elementare, vi apparirà chiaro che, nell'*output stream* supportato dalla finestra del terminale, se non trovate un modo VOI per imporlo, le andate a capo (i caratteri '\n') NON SONO inserite alla stessa distanza l'una dall'altra (e anche per quale motivo ciò accade...).

# include <iostream>  
using namespace std;  
  
int main(  )  
{  
for(int i=0; i <= 10; ++i)  
cout << i/3. << '\n' << 2\*i/3. << '\n' << i << '\n';  
}

**Canto diciannovesimo: il sistema ios; parte I: breve introduzione agli OGGETTI**

Questo canto è solo un'anticipazione strettamente indispensabile di quanto sarà approfonditamente ripreso più avanti, fatta per poter parlare di un sistema di INPUT/OUTPUT basato sugli OGGETTI, quale è il sistema *ios* del C++, sapendo dare un significato a ciò che si scrive e legge.  
D'altronde delle due una: O si parla subito compiutamente degli OGGETTI, ma senza poter fare il minimo cenno a un po' di INPUT/OUTPUT evoluto, che non sia lo sfruttamento banale ed esclusivo di *cin* e *cout*, come si è sempre fatto finora, O si affronta subito compiutamente *ios*, ma avendo per forza di cose una conoscenza solo approssimativa, e finalizzata allo scopo, di che cosa sia un oggetto.  
  
Scelta questa seconda opzione, diremo solo, per ora, che un OGGETTO è assimilabile a una variabile e quindi, come tale, va per prima cosa DICHIARATO.   
  
Dichiarare un oggetto ha il consueto significato, ossia quello di attribuirlo a un TIPO (o, se preferite, attribuirgli un TIPO).  
  
La differenza è che il **TIPO STESSO DEVE ESSERE STATO A SUA VOLTA PREVENTIVAMENTE DICHIARATO E DEFINITO**, perché altrimenti il compilatore **NON LO CONOSCE** (ricordatevelo: lui conosce SOLO i suoi tipi nativi, per ciascuno dei quali ha addirittura una parola RISERVATA nel vocabolario).  
  
Come si dichiara e definisce il tipo cui appartiene un oggetto è materia rimandata all'approfondimento cui si è accennato; tanto, per quanto riguarda *ios*, CI PENSANO, a vostro vantaggio, I DOCUMENTI CHE INCLUDERETE nel vostro programma. A voi tocca quindi sapere solo questo: che quei documenti inclusi vi mettono appunto a disposizione UNA DOVIZIA DI TIPI, già dichiarati e definiti, con cui potrete appunto DICHIARARE i vostri oggetti secondo necessità (vale a dire che ci sarà il TIPO adatto per ogni evenienza, e voi dichiarerete quindi un oggetto di quel tipo).  
  
Non solo; il documento *iostream* (che abbiamo SEMPRE incluso, dal momento immediatamente successivo al *big bang*) fa anche di più: VI DICHIARA PURE gli oggetti *cin* e *cout*, ciascuno del tipo giusto (per i più curiosi si tratta, rispettivamente, dei tipi chiamati *basic\_istream<char>* e *basic\_ostream<char>*; ma questo, al momento attuale, potete dimenticarlo appena letto) e VI APRE PERFINO i due *stream* che gestiscono: rispettivamente un *input stream* e un *output stream*, che, proprio per come sono aperti (NON da VOI), sono chiamati *stream standard*.  
Ciò è TALMENTE VERO che si è sempre fatto dell'INPUT/OUTPUT (appunto *standard*) SENZA AVER DOVUTO FAR NULLA di quel che si imparerà a fare da ora in avanti.  
  
Un oggetto dichiarato nasce, vive e muore nell'ambito in cui si trova la sua dichiarazione e in tutti gli ambiti ad esso interni: *cin* e *cout* sono dichiarati nel *namespace std*, a sua volta situato nell'ambito GLOBALE. Per questa ragione sono fruibili OVUNQUE al solo prezzo dell'inserimento della linea *using namespace std;* (QUESTO ERA GIÀ stato detto fin dall'inizio, ma *repetita iuvant*).  
  
Da questo punto di vista, quindi, è, ancora, come qualsiasi altra variabile, tanto che qualsiasi altra variabile, anche una volgare variabile intera, può essere chiamata, a buon diritto, un OGGETTO del tipo *int*.  
  
Diversamente dalle variabili/oggetti appartenenti ai tipi nativi, però, un oggetto appartenente a un tipo dichiarato e definito dal programmatore o, comunque, al di fuori del vocabolario del compilatore, può essere, e quindi quasi sempre **È**, un **AGGREGATO** di variabili/oggetti e di funzioni che l'oggetto PORTA IN SÉ e che sono accessibili al programmatore attraverso l'operatore . (se non si legge è un punto).  
  
Notate che c'è scritto variabili/oggetti VOLUTAMENTE, nel senso che di tale AGGREGATO possono far parte anche oggetti che siano a loro volta degli AGGREGATI; e c'è scritto, ALTRETTANTO VOLUTAMENTE *funzioni* perché è VERO: dell'AGGREGATO fanno parte anche vere e proprie funzioni le quali (MERAVIGLIA!) conoscono i valori di tutte le variabili/oggetti dell'aggregato SENZA BISOGNO che vengano loro trasmessi in alcun modo, così che gli eventuali parametri che venissero loro trasmessi sono, per natura, estranei all'aggregato (non significa che sia vietato trasmettere variabili dell'aggregato: significa che sarebbe una perdita di tempo).  
  
Se siete dotati di quella virtù propria del buon programmatore denominata FANTASIA, dovreste già aver pronunciato nella vostra mente il digramma GL (Gran Linguaggio), se non altro come abbreviativo del fumettistico *GLub* della deglutizione, e dovreste già intravedere i vastissimi, sconfinati, orizzonti che si schiudono a tutte le vostre virtù. Se non li vedete restate tranquilli: non è detto che non abbiate fantasia; probabilmente è solo poco allenata o, nella peggior ipotesi, ottusa dalle playstation o dalle cuffiette.  
  
Ovviamente (c'era dubbio?) si possono anche dichiarare puntatori a oggetti: in tal caso, per accedere a qualsiasi componente dell'aggregato, l'operatore . (punto) DEVE essere sostituito con l'operatore *->* (un segno meno seguito immediatamente dal segno di maggiore), chiamato familiarmente operatore freccia. Se tornate ADESSO a leggere il canto sugli operatori, troverete che questi due erano stati rimandati al futuro, che ORA è giunto.  
  
Per calarsi nella concretezza dell'esempio, ecco due espressioni, entrambe corrette, per richiedere che venga eseguita la funzione *get(  )*, facente parte dell'aggregato dell'oggetto *cin*:

cin   .  get(  );  
(&cin)   ->  get(  );

Il motivo della presenza delle parentesi attorno a *&cin*, necessarie per non incorrere in errore, dovrebbe esservi ormai noto e chiaro, così come la discrezionalità delle spaziature attorno ai due operatori, qui inserite solo per rendere agevole la lettura.  
  
Per concludere questo canto introduttivo sugli oggetti, si anticipa un'altra loro caratteristica FORMIDABILE: tra le funzioni che, come si è detto, possono far parte dell'aggregato, o essergli comunque strettamente collegate (vedremo come), ve ne sono alcune la cui esecuzione, qualora il programmatore o chi per lui/lei le DEFINISCA, è invocata AUTOMATICAMENTE, e AL MOMENTO GIUSTO, dal compilatore stesso, SENZA che il programmatore DEBBA PIÙ PENSARCI (non significa che non debba esserne consapevole, perché comunque è lui/lei ad averle scritte).   
  
Tra queste ve ne sono che possono stravolgere completamente il significato di un operatore e che sono appunto ESEGUITE AUTOMATICAMENTE quando l'oggetto che le detiene diventa operando sinistro (e sottolineo APPOSTA sinistro) di quell'operatore, se trattasi di operatore binario, o comunque operando dell'operatore, se trattasi di operatore unario.  
  
È quello che è SEMPRE tacitamente accaduto OGNI VOLTA che si è scritto *cin >> x;* oppure *cout << x;*:  
  
in queste espressioni vengono DI FATTO ESEGUITE due funzioni appositamente definite PER gli aggregati *cin* e *cout* rispettivamente, e che hanno il compito di modificare completamente il significato originario dei due operatori *>>* e *<<*, che è, non dimenticatelo, quello di scorrimento dei *bit* di un intero nei due sensi, e che quindi NULLA C'ENTREREBBE con operazioni di lettura o scrittura da/su *stream*. I valori che quelle funzioni restituiscono, come si era anticipato senza spiegazioni quando si era parlato di espressioni, sono, rispettivamente, PROPRIO *cin* e *cout* medesimi, il che spiega appunto la ricorsività delle operazioni di lettura e scrittura compiute tramite questi operatori, ossia espressioni come *cin >> x >> y;*.  
  
La vostra fantasia, a questo punto, dovrebbe galoppare più velocemente di Bucefalo, Xanto, Balio e Pegaso messi in quadriga e la vostra curiosità (virtù, questa, propria più del Fisico che del Programmatore) dovrebbe aver raggiunto livelli da parossismo. Se invece avete la curiosità di una carota o di una vongola, e la vostra fantasia perde per distacco da Ronzinante...forse avete sbagliato mestiere.  
  
Da questo momento i termini variabile e oggetto saranno usati con lo stesso significato e le funzioni che appartengono a un oggetto saranno frequentemente denominate *metodi dell'oggetto*.

**Canto ventesimo: il sistema *ios* per l'INPUT**

Il sistema *ios* è vasto e intricato come le catacombe romane: in questo canto (e nel successivo, per gli *stream* di *output*) se ne darà una rassegna, di certo non esaustiva e non orientata agli scavi in profondità che mettano in luce i dettagli fini del suo funzionamento, ma comunque adeguata all'utilizzo proficuo da parte del buon programmatore.   
  
Come già si è detto, un programma, per il solo fatto di iniziare a essere eseguito, ha un *input stream*, il cosiddetto *standard input stream*, aperto e disponibile sotto la gestione dell'oggetto *std::cin* (o semplicemente *cin* se si è inserita la linea *using namespace std;*) che, salvo avviso contrario, ha per supporto fisico la tastiera.  
  
L'avviso contrario cui si è accennato consiste nella proprietà di reindirizzamento dello *standard input stream* contemplata nel sistema operativo Linux e nei sistemi operativi simil-Unix, o anche nel rimpianto DOS; se si manda in esecuzione il proprio documento eseguibile (*a.out*) col comando

./a.out   <  un\_certo\_documento

in cui *un\_certo\_documento* è il percorso, appunto, di un certo documento situato da qualche parte nel disco fisso, QUEL documento diventa automaticamente il supporto fisico dello *standard input stream*, AL POSTO DELLA TASTIERA, da cui il programma non leggerà più NULLA, e lo *stream* così reindirizzato sarà comunque gestito dall'oggetto *cin* senza cambiare il codice in ALCUN MODO.  
  
Questa caratteristica del sistema operativo (NON del linguaggio, cui non interessa NIENTE quale sia il supporto fisico di uno *stream*, se non per farlo gestire dall'oggetto giusto, ricordatevelo) è di incomparabile pregio specialmente durante la fase di ricerca di errori commessi da un programma: immaginatevi solo quanta possa essere la noia di dover ripetere la digitazione di non si sa quanti dati (ma anche di UNO SOLO!) a OGNI riesecuzione del programma dopo aver fatto le correzioni auspicabilmente in grado di individuare e uccidere il baco...piuttosto che scrivere UNA TANTUM un documento contenente ESATTAMENTE TUTTI I DATI che dovrebbero essere digitati sulla tastiera, NELLO STESSO ORDINE, e usare il comando sopra riportato.  
Naturalmente il nome del documento sarà più corto, ovvero si useranno gli strumenti che la *shell* mette a disposizione per ripetere un comando SENZA DOVERLO RISCRIVERE (GRAN SISTEMA OPERATIVO!).   
  
Tuttavia è chiaro che devono esserci altri modi per rendere supporti di *input stream* dei documenti su disco, se non altro per poterne gestire più di UNO SOLO contemporaneamente, così come per avere supporti di *input stream* di altra natura. Diremo quindi ADESSO che cosa occorre per avere supporti su disco diversi dallo *standard input stream* e per avere supporti in memoria.  
  
Per prima cosa occorrerà includere gli appropriati documenti che DICHIARANO I TIPI necessari, come detto tre canti indietro, ossia inserire le linee

# include <fstream>  
# include <sstream>

in AGGIUNTA e SUCCESSIVAMENTE alla consueta *#include <iostream>*, la prima per i supporti su disco e la seconda per i supporti in memoria. Dato che l'inclusione di un documento di sistema NON HA controindicazioni si potrebbero includere anche quando non servano.  
  
A questo punto OCCORRE DICHIARARE un oggetto PER OGNI *input stream* da disco che si voglia tenere SIMULTANEAMENTE APERTO e un oggetto PER OGNI *input stream* da memoria che si intenda utilizzare simultaneamente. Ecco le dovute dichiarazioni, in cui si possono apprezzare i nomi dei TIPI dichiarati nei documenti inclusi (si sottintende l'uso di *using namespace std;*, altrimenti i nomi dei tipi dovrebbero essere completati facendoli precedere da **std::**):

**ifstream**    input\_stream\_da\_disco;  
**istringstream**   input\_stream\_da\_memoria;

I nomi degli oggetti sono volutamente un autocommento del loro scopo: in casi concreti sarà opportuno che siano più corti, dato il gran numero di volte che appariranno nel codice.  
I nomi dei TIPI, peraltro, sono QUELLI, modulo l'uso eventuale della parola di vocabolario *typedef* (non andatela a leggere, adesso), e, a parte la desinenza comune *stream*, reminiscenza di quel che sono, significano che si tratta di *stream* di INPUT (la lettera *i* iniziale) col supporto indicato dal resto del nome: *f* è l'iniziale di *file*, mentre *string* indica che il supporto in memoria in realtà è una stringa, che a sua volta è una collezione ordinata di *bytes*, disposti a fungere da *stream*; in fondo uno *stream* è esattamente la stessa cosa...per definizione.  
  
Le precedenti dichiarazioni possono essere sostituite da dichiarazioni di puntatori, opportunamente inizializzati, o *array*, come per qualsiasi altro tipo di variabili:

// dichiarazioni alternative  
**std :: istringstream**  
   array\_di\_input\_stream\_da\_memoria[3];  
**std :: ifstream**  
   \*puntatore\_a\_input\_stream\_da\_disco =   
new **std :: ifstream**;

Le dichiarazioni, pur indispensabili, restano fini a sé stesse finché lo *stream* non viene APERTO. Per questa bisogna i due tipi in questione dispongono di metodi, ossia di funzioni definite nell'aggregato che descrivono, DIVERSI e DISTINTI; usando i nomi citati nel primo esempio di dichiarazione, ecco come si effettua l'apertura dei rispettivi *stream*:

input\_stream\_da\_memoria   .   str(" 10 20 30 ");  
input\_stream\_da\_disco   .   open("nome\_di\_documento");

L'uso dell'operatore punto è già stato introdotto; i parametri trasmessi ai metodi *open* e *str* sono stati qui volutamente indicati come delle costanti per evidenziarne il tipo senza che occorresse dichiararli; nella pratica si trasmetteranno variabili opportunamente dichiarate e inizializzate. I valori delle costanti ne commentano da soli il significato: il metodo *open* realizza, se ci riesce, la connessione tra l'*input stream* gestito dal proprio oggetto e il documento chiamato effettivamente PROPRIO *nome\_di\_documento* (questo **È** il difetto delle costanti) che dovrebbe trovarsi nella stessa cartella in cui risiede il documento eseguibile, e il cui contenuto va a costituire il supporto dello *stream*; il metodo *str*, a sua volta, realizza la stessa connessione tra il proprio oggetto e la stringa trasmessagli, la quale risiede nella memoria del programma stesso, come del resto anche quella trasmessa a *open*, e costituisce LEI il supporto dello *stream*.  
  
Si sottolinea che, mentre il metodo *str*, così com'è invocato, non può fallire, il metodo *open*, invece, fallisce miseramente se NON ESISTE nella cartella corrente un documento chiamato *nome\_di\_documento*, oppure se tale documento esiste, ma è inaccessibile in lettura all'utilizzatore del programma: un documento privato appartenente a qualcun altro; il calcolatore è estremamente educato e rispettoso della cosiddetta *privacy*. Tale situazione, da doversi evitare da parte del buon programmatore, può essere comunque recuperata andando a controllare il buon esito dell'operazione e prendendo provvedimenti nel caso che il buon esito non sia tale. Quando ciò accadesse, l'oggetto *input\_stream\_da\_disco* verrebbe booleanamente falsificato da *open* stessa, diventando praticamente inservibile, ma nello stesso momento acquisterebbe valenza un'istruzione come

if(!input\_stream\_da\_disco) {  
cout <<"documento non valido: facciamo qualcosa\n";  
// omissis  
}

seguita magari da un bel *throw* di qualche tipo o da qualsiasi altra iniziativa utile a non far terminare il programma al primo tentativo di lettura da uno *stream* non validamente aperto.  
  
La dichiarazione e l'apertura di un *input stream* (di qualunque *stream*) possono essere unificate nel seguente modo:

**istringstream**  
input\_stream\_da\_memoria(" 10 20 30 ");  
**ifstream**   input\_stream\_da\_disco("nome\_di\_documento");

in cui l'invocazione dei metodi *str* e *open* è stata equivalentemente sostituita da una nuova forma di inizializzazione contestuale con la dichiarazione. Se tornate indietro a rileggere il canto sulle inizializzazioni troverete che vi era stato detto che ne esistevano molte forme e che in quel canto primordiale se ne sarebbero introdotte solo due: quella per assegnamento e quella per lettura; strada facendo si era poi incontrata anche l'inizializzazione immediata degli elementi di un array tramite gli inizializzatori delimitati da graffe, con o senza =, e questa, nel canto presente, è un'ulteriore forma di cui comprenderete molto presto la profondità del significato.  
  
Qualcuno potrebbe obiettare: *"ma allora i metodi* ***open*** *e* ***str*** *sono del tutto inutili; perché si dovrebbe perder tempo a usarli, quando è contemplata quest'ultima metodologia?"*  
  
Risponderòvvi con la seguente imperitura terzina del Sommo Poeta:  
  
*O insensata cura de' mortali,  
quanto son difettivi silogismi  
quei che ti fanno in basso batter l'ali!  
(Par. XI, 1-3)*In effetti basterebbe anche una risposta più prosaica: senza quei metodi OGNI oggetto dichiarato potrebbe essere usato SOLAMENTE per un UNICO supporto: quello che appare nella dichiarazione inizializzata; mentre, grazie a QUEI metodi, e al fratellino di *open* denominato, ma guarda, *close*, il seguente segmento di programma mostra, in modo autocommentato, come lo STESSO OGGETTO possa essere riciclato quante volte si voglia, purché NON SI ESEGUA *open* su uno *stream* che fosse già aperto (*à quoi bon?* Ciò rovinerebbe lo *stream* proprio come se si tentasse *open* su un documento inesistente, ed è per questo che esiste *close*):

ifstream  
input\_stream\_da\_disco("nome\_di\_documento");  
// si sprema lo *stream* fino all'ultima goccia...  
input\_stream\_da\_disco . close(  );  
// via "nome\_di\_documento"; non mi servi più  
// .... ma input\_stream\_da\_disco   
// è TUTTORA dichiarato....  
// (non siamo mica usciti   
// dall'ambito della dichiarazione)  
// e allora ....   
input\_stream\_da\_disco  
. open("nome\_di\_un\_altro\_documento");  
// se non ci fosse *open* sarebbe stato necessario  
// dichiarare un altro oggetto, lasciando il precedente  
// inutilizzato a consumare risorse...

Un *input stream* APERTO, comunque lo sia stato, è, abbiamo detto, disponibile a fornire al programma i dati che supporta fino a quando

* per un *ifstream* si invoca il metodo *close(  )*: per avere di nuovo uno *stream* disponibile occorre, come si è appena visto, eseguire novamente *open*;
* per un *istringstream* si ri-invoca il metodo *str*: e a questo punto sarà disponibile l'*input* dalla nuova stringa trasmessa a quel metodo;
* si incorre in errore, di qualsiasi natura, durante un'operazione di lettura: in tal caso NESSUN metodo orientato alla presa dati funziona più, fino a quando non si esegua il metodo *clear(  )* che cancella la segnalazione di allarme impostata dall'oggetto nelle proprie variabili interne.

Ora si inizierà a imparare a destreggiarsi *lungo lo stream*, come si impara a guidare una canoa su un fiume o torrente per andare a caccia di trote... volevo dire...di *bytes*.  
  
Tutti sapranno che per guidare una canoa ci vuole la pagaia, a una o due pale; questa consente di arrivare in qualunque punto accessibile del torrente, senza pescare (ancora) nulla. L'equivalente della pagaia, in un *input stream*, è il metodo *seekg*, che attende UNO o DUE argomenti (significa che ne esistono due versioni in *overload*, ricordate?): il primo, o unico, argomento, di tipo riconducibile a intero segnato adeguatamente modificato, serve a dire al metodo per quanto pagaiare, ossia di quanti *bytes* spostarsi lungo il torrente (lo *stream*); se viene fornito questo unico parametro, il cui tipo è stato dichiarato nei documenti inclusi e si noma, per esteso, *std::streamsize*, o anche *std::streampos*, o anche *std::pos\_type*, o anche... MA BASTA! ... dicevo che quando viene fornito SOLO LUI, il suo valore si interpreta a partire dall'inizio dello *stream*, ossia dove ci si trova quando lo *stream* è stato appena aperto come si è fatto in questo canto. Se invece sono forniti tutti i due parametri, il valore del primo si interpreta a partire dal valore del secondo parametro; quest'ultimo può essere SOLO una di queste tre variabili già dichiarate nell'ambito indicato esplicitamente:

* *ios :: beg*  
  il punto di partenza della pagaiata è collocato all'inizio dello *stream*; fornire questo valore equivale a NON fornire il secondo parametro;
* *ios :: end*  
  il punto di partenza della pagaiata è collocato alla FINE dello *stream*;
* *ios :: cur*  
  il punto di partenza della pagaiata è collocato nel punto dello *stream* in cui ci si trova nel momento in cui il metodo viene eseguito;

**Osservazioni in corso d'opera:**

1. la prima bella notizia è che il metodo *seekg*, data l'universalità di un *input stream* APERTO, ce l'hanno, IDENTICO, tanto gli *ifstream* quanto gli *istringstream*;
2. in un *input stream* come quelli di cui si sta parlando non si può pagaiare oltre la fine né prima dell'inizio dello *stream*: se ciò accadesse la canoa s'impantana, ossia si incorre in una condizione di errore e lo *stream* si guasta;
3. in un *input stream* come quelli di cui si sta parlando, se si trasmette a *seekg* il secondo parametro col valore *ios::end*, il primo parametro deve avere valore NON POSITIVO, altrimenti si cade nel punto precedente. In altri termini, come è ovvio, se si parte da *ios::end*, si può solo pagaiare contro corrente;
4. considerazioni equivalenti concernenti il valore del primo parametro quando il secondo è dato come *ios::beg* o *ios::cur* sono lasciate, alla luce di quanto scritto al punto 2., al buon senso di chi legge;
5. in ogni momento è possibile conoscere il valore attuale dell'indice di lettura, ossia la posizione in cui la canoa è ancorata lungo lo *stream*; questo è il valore, dello stesso tipo del primo argomento di *seekg*, che restituisce al chiamante il metodo *tellg(  )*. In altre parole il valore dell'espressione *input\_stream\_da\_disco.tellg(  );* è quello della variabile *ios::cur* IN QUEL MOMENTO;
6. dovrebbe essere ovvio (il metodo esiste APPOSTA) che un'esecuzione ben riuscita del metodo *seekg* comporta il coerente aggiornamento del valore dell'indice di lettura, ossia del valore corrente della variabile *ios::cur* stessa: la canoa getta l'ancora nel punto raggiunto al termine della pagaiata;
7. se si volesse dichiarare una propria variabile che fosse legittimata a essere inizializzata col valore di una delle tre variabili dichiarate, il tipo da usare nella dichiarazione sarebbe, per esteso,   
   *std :: ios\_base :: seekdir*
8. né *ios* né *ios\_base* sono dei *namespace* come invece è *std*, anche se appaiono usati con la stessa sintassi; in un futuro prossimo si comprenderà la differenza: per adesso, che non vi salti in mente una furbata del tipo *using namespace ios;* con l'intento di potersi risparmiare di scrivere *ios::* davanti al nome delle variabili, perché sarebbe un ERRORE;
9. anche l'oggetto *cin* ha il metodo *seekg* (è pur sempre un *input stream*), ma quando il suo supporto è la tastiera non se ne può apprezzare la funzionalità: in effetti la tastiera è una sorta di rapida del torrente e lì ci vorrebbe un salmone, non una canoa. Tuttavia il metodo torna perfettamente fruibile in caso di reindirizzamento dello *standard input stream* da documento su disco.

Ora che si sa come muoversi lungo il fiume è giunto il momento di pescare, sapendo che ciò che si pescherà si trova a partire dalla posizione attuale lungo lo *stream*, ossia dal valore corrente dell'indice di lettura, in poi.  
  
La più semplice operazione di lettura è quella che È SEMPRE STATA COMPIUTA FIN QUI, senza domandarsi come funzionasse, e che utilizza l'operatore di estrazione *>>*, disponibile per QUALSIASI *input stream* correttamente aperto: si tratta di un'operazione di lettura riservata a supporti testuali, ossia a *input stream* i cui dati contenuti siano leggibili e comprensibili ANCHE DA OCCHIO UMANO. Ad esempio l'oggetto di tipo *istringstream* dichiarato sopra, supportato dalla stringa costante *" 10 20 30 "*, ha chiaramente un supporto testuale, dato che anche un bambino delle elementari sa leggerne il contenuto. Pertanto l'*input stream* associato consente l'uso dell'operatore di estrazione e i dati possono quindi essere letti nella stessa maniera in cui si leggerebbero se fossero digitati sulla tastiera. Ciò non significa che il compilatore segnali errore se si usa l'estrattore su un supporto non testuale, perché il compilatore NON PUÒ conoscere la natura del supporto; significa semplicemente che usare l'estrattore su supporti NON TESTUALI comporterà molto verosimilmente **gravissimi malfunzionamenti del programma**.  
  
Quando l'operatore di estrazione *>>* si trova sulla destra di un oggetto che sia un *input stream* di qualunque tipo, o se si preferisce: quando un tale oggetto è operando sinistro di *>>*, come si è già anticipato viene automaticamente eseguita una funzione strettamente legata all'aggregato costituito dall'oggetto stesso; a tale funzione viene automaticamente trasmesso come parametro l'operando destro di *>>* e quindi quel che la funzione farà dipenderà dal TIPO di quel parametro. Quelli di voi che siano stati più attenti avranno forse intuito che esisteranno numerose versioni di questa funzione, in *overload*: una per ogni possibile tipo nativo; e che ognuna di queste funzioni **dovrà ricevere il parametro trasmesso prefissandolo con &**.  
  
Tutto ciò premesso, ecco come agisce *>>*, in ordine cronologico:

1. partendo dalla posizione attuale nello *stream* si procede in avanti superando e **IGNORANDO** TUTTE le eventuali occorrenze di *bytes* di spaziatura testuale: spazi veri e propri, tabulazioni orizzontali e verticali, andate a capo; ecco perché bisogna che il supporto sia testuale. Durante questo avanzamento viene tenuto costantemente aggiornato l'indice di lettura: se si oltrepassa la fine dello *stream* viene segnalato errore e la funzione fallisce.
2. se si è superato il punto precedente, o se non è stato affatto necessario compierlo, significa che nella posizione attuale dello *stream* NON SI TROVA un *byte* di spaziatura testuale; allora si estraggono tanti *bytes* quanti sono quelli che si trovano successivamente e che siano interpretabili come legittimamente appartenenti a un dato del tipo dell'operando destro di *>>*, mantenendo sempre aggiornato l'indice di lettura; l'avanzamento nello *stream* cessa quando o si incontra un *byte* non rispondente alle caratteristiche richieste, e che NON viene estratto dallo *stream*, o si supera la fine dello *stream*: in quest'ultima evenienza viene segnalato errore e la funzione fallisce.
3. se si è superato il punto precedente si conta il numero di *bytes* che sono stati estratti: se è zero la funzione segnala errore e fallisce; altrimenti trasmette i *bytes* estratti alle *routines* di conversione interna che li trasformano nella rappresentazione binaria del dato da leggere, conforme all'architettura della macchina ospite, e lo memorizzano nella locazione di memoria ricevuta come argomento, completando di fatto la lettura del dato. A questo punto la funzione restituisce al chiamante l'oggetto stesso che l'ha eseguita, rendendo immediatamente possibile una nuova (eventuale) lettura dallo stesso *stream* (GRAN LINGUAGGIO).
4. per quanto detto ai punti precedenti la funzione può fallire per DUE ragioni: o perché il tentativo di lettura provoca la violazione della FINE dello *stream* (vedi punti 1. e 2.) o perché nello *stream* NON SI TROVA UNO STRACCIO DI *BYTE* idoneo al tipo di dato ricevuto (vedi punto 3.). In ogni caso di fallimento lo *stream* risulta compromesso e, in assenza di rimedi previsti dal programmatore, diventa inutilizzabile: quanto al valore del dato che si pretendeva di leggere, dipende dal tipo di dato e dal tipo di errore, ma comunque si tratta di un valore su cui non è possibile fare ALCUN AFFIDAMENTO.

Si dà ora, come esempio didattico, un breve codice che illustra SOLO quanto appreso finora, e siccome avete ancora MOLTO da apprendere, questo vi basti a capire che i programmi VERI non si scriveranno così. Utilizzeremo come *input stream* campione un oggetto di tipo *istringstream* tenendo ben presente che TUTTO funzionerebbe INTATTO per qualsiasi altro tipo, cambiando SOLTANTO la dichiarazione e il metodo di apertura.

# include <iostream>  
# include <sstream>  
using namespace std;  
  
int main(  )  
{istringstream is("10 + 20 = 30 ");  
int n[2], errore = -1;  
double x;  
char c[2];  
ios\_base :: seekdir d = ios :: beg;  
cout  
<<"appena aperto lo stream si trova prima del byte " <<is . tellg(  )<<'\n';  
is . seekg(0, ios :: end);  
streamsize s = is . tellg(  );  
cout << "lo stream contiene " << s << " byte\n";  
is . seekg(0, d);  
is >> n[0];  
cout << "dopo aver letto " << n[0]  
<< " lo stream è prima del byte "  
<< is . tellg(  ) << '\n';  
d = ios :: end,  
is . seekg(-3, d),  
cout   
<< "dopo seekg(-3, ios::end) è prima del byte "   
<< is . tellg(  ) << '\n';  
is >> x;  
cout  
<< "dopo aver letto " << x   
<< " è prima del byte " << is . tellg(  )  
<< " che coincide con " << s-1 << '\n';  
d = ios :: cur,  
is . seekg(-4, d),  
cout   
<< "dopo seekg(-4, ios::cur) è prima del byte "  
<< is . tellg(  ) << '\n';  
is >> c[1];  
is . seekg(-6, d),  
cout << "dopo aver letto " << c[1]   
<< " e seekg(-6, ios::cur) è prima del byte "  
<< is . tellg(  ) << '\n';  
is >> c[0] >> n[1];  
cout   
<<"eseguite le ultime letture utili"  
<<   
" si commette apposta un errore: oltrepassare l'inizio.\n";  
is . seekg(-20, d);  
is >> errore;  
if(!is) cout << "stream rovinato; errore non letto "  
<< errore << '\n';  
cout << "in base a quanto letto risulta "   
<< n[0]<<c[0]<<n[1]<<c[1]<<x<<'\n';  
}

**ESEGUITE** questo programma, che vi spiegherà da solo tutto il precedente elenco in quattro punti, senza bisogno di aggiungere altro.  
  
Se proprio avete bisogno che vi si aggiunga qualcosa eccovi alcuni commenti sul codice:

* nella stringa che fa da supporto all'*input stream*, se si pretende di poter leggere correttamente anche l'ultimo dato che contiene, occorre che, prima delle virgolette di chiusura, che costituiscono la FINE dello *stream*, ci sia ALMENO UNO SPAZIO, ex punto 2. dell'elenco, come in effetti è stato fatto;
* i dati numerici inseriti nella stringa sono idonei a essere interpretati, come fa il programma, sia come pertinenti ai tipi riconducibili a *int* sia ai tipi *float* e *double*. Per questi ultimi due tipi sarebbero stati considerati pertinenti anche tutti i *bytes* che concorrono a scrivere un numero reale in notazione esponenziale, nell'ordine corretto, come ad esempio in *12.37e-2*;
* l'operando destro di *>>*, ossia il parametro trasmesso alla funzione sottintesa, non può MAI essere un puntatore, con la SOLA eccezione dei puntatori a *void* e a *char*.  
  In effetti, se si provasse a scrivere in QUELLA posizione un puntatore a qualsiasi altro tipo nativo, il compilatore emetterebbe una filza di diagnostici d'errore addirittura imbarazzante (provate e crederete), mentre se vi si pone un puntatore a *void* o a *char* resta impassibile e compila senza discutere;
* a chiunque dovrebbe sorgere spontanea la domanda sul motivo di questa differenza di trattamento: in nessun caso, ricordiamocelo, avrebbe senso ritenere possibile che il valore di un puntatore possa essere letto da uno *stream* esterno al programma; dunque appare plausibile che il compilatore rifiuti la presenza di un puntatore in una posizione in cui, in genere, si trova qualcosa che dovrebbe essere letto...e allora perché accetta puntatori a *void* o a *char*? che cosa viene letto, se non può essere letto un valore per il puntatore che si trova lì? In realtà il valore di un puntatore a *void* può essere letto, ma il significato di tale operazione esula dal contesto presente e riguarda solo situazioni molto rare e avanzate; tuttavia la domanda permane a proposito dell'altro tipo di puntatore...
* la risposta è abbastanza semplice: i *char* che quel puntatore punta; lo *stream* viene fatto avanzare estraendone qualsivoglia *byte*, ex punto 2. dell'elenco puntato, fino, ed escluso, al primo *byte* di spaziatura, ovvero, con l'errore implicato, fino allo sfondamento della FINE dello *stream*; e può farlo solamente per i puntatori a *char* perché il criterio di fermarsi alla spaziatura è abbastanza plausibile per decretare la fine di una stringa di caratteri, ma non affatto per decidere che subito dopo non possa esserci, tanto per dire, un altro *int* che dovrebbe essere puntato anche lui. E il compilatore, in assenza di criteri univoci, non assume responsabilità e segnala ERRORE; va notato, per concludere, che quando cessa l'estrazione dallo *stream* dei *bytes* che dovranno essere puntati dal puntatore a *char* in corso di lettura, a quelli effettivamente estratti ne viene sempre, e sistematicamente, aggiunto UNO, di valore NULLO, che è denominato terminatore della stringa.
* la precedente risposta, tuttavia, fa sgorgare dal petto ALMENO ALTRE TRE DOMANDE, sempre ammesso che non siate quella cozza di cui si è parlato; o era una vongola?
  1. il puntatore a *char* trasmesso a *>>* deve essere già inizializzato o ci pensa la funzione, visto che lo riceve prefissato con &?
  2. in caso di risposta affermativa alla domanda precedente, come si fa a predire quanti *char* debba essere predisposto a puntare?
  3. ...e se si volesse che i *bytes* di spaziatura entrassero anche loro a far parte dei *char* puntati, magari assieme ad altri *char* successivi a uno o più spazi?

Risposte alle tre ultime domande:

1. sì, il puntatore a *char* deve già essere stato inizializzato: la funzione NON si assume responsabilità di allocazioni di memoria al suo interno, con effetti sul chiamante;
2. in teoria un modo ci sarebbe: fare una pre-scansione dello *stream* alla ricerca delle posizioni in cui si trovano le spaziature...ma siccome si tratta di una maniera più dispendiosa, sia di tempo sia di risorse, rispetto a una semplice sovrastima ragionevole, si preferisce DI GRAN LUNGA quest'ultima opzione; si sottolinea che, quand'anche la predetta sovrastima non fosse adeguata, vengono COMUNQUE estratti dallo *stream* TUTTI i *bytes* competenti e TUTTI, con in più il *byte* terminatore, sono memorizzati all'indirizzo del puntatore; se questo è insufficiente, il programma si interromperà eventualmente in modo prematuro NON a causa di errori nella gestione dello *stream*, ma piuttosto per violazione degli accessi in memoria.  
   Programmatore avvisato, mezzo salvato...
3. in tale evenienza, molto semplicemente, NON si usa l'estrattore: ci sono molte altre maniere disponibili...tra le quali non sarà difficile scegliere quella utile alla bisogna.

Esaurita la trattazione del comportamento base dell'operatore *>>*, si passerà adesso a presentare tutti, o quasi, i metodi preposti alle operazioni di lettura cosiddette non formattate o anche binarie, ossia quelle che agiscono su supporti generalmente non accessibili all'esperienza sensoriale dell'*homo sapiens sapiens*.  
In tali supporti NON ESISTONO *bytes* di spaziatura che vengano IGNORATI, vale a dire, in definitiva, TRATTATI DIVERSAMENTE DA ALTRI *BYTES*, come abbiamo visto accadere nei supporti testuali: insomma, ogni *byte* è potenzialmente equivalente a ogni altro e nessuno vanta particolari diritti di esclusiva.  
  
Si inizia con due metodi, per così dire, ibridi, nel senso che, pur essendo orientati alla lettura di supporti non formattati, possono essere utilizzati proficuamente anche su supporti testuali: si tratte dei metodi *get* e *getline* che sono definiti con diversi *overload* e/o argomenti *standard* come qui appresso specificati:  
  
*overload* del metodo *get*:

1. *get(  )*
2. *get(char&)*
3. *get(char\*, int, char = '\n')*
4. *// altri overload, pur definiti, sono qui taciuti.*

dichiarazione del metodo *getline*:  
  
*getline(char\*, int, char='\n')*

Appare subito evidente che la dichiarazione di *getline* è IDENTICA all'*overload* numero 3 di *get*, e quindi ci dovrà essere una differenza nel comportamento. Iniziando appunto da *getline* il significato degli argomenti è il seguente:  
a partire dalla posizione attuale dell'*input stream* ne vengono estratti in sequenza *bytes* di qualsiasi valore (significa: compresi quelli di spaziatura), che sono ordinatamente memorizzati a partire dall'indirizzo specificato dal puntatore trasmesso come primo argomento, e che deve essere sufficientemente inizializzato, fino a quando uno di questi tre controlli, effettuati dalla funzione nell'ordine qui appresso specificato, dà esito positivo:

1. a furia di estrarre *bytes* si incoccia nel superamento della fine dello *stream*; in questo caso, ovviamente, la funzione attiva coerentemente il sistema di allarme;
2. nell'*input stream* si incontra il *byte* fornito come terzo argomento, ossia *'\n'* se tale argomento non è fornito; in questo caso tale *byte* è l'ultimo a essere ESTRATTO dall'*input stream*, con aggiornamento coerente del valore dell'indice di lettura, MA NON VIENE AGGIUNTO a quelli indirizzati dal primo argomento, vale a dire che finisce PERSO. La funzione aggiunge ai *bytes* indirizzati dal primo argomento il *byte* terminatore NULLO e si conclude felice e contenta;
3. dall'*input stream* sono già stati estratti un numero di *bytes* pari a UNO IN MENO di quanti ne indica il valore del secondo argomento fornito: la funzione aggiunge il *byte* terminatore a quelli indirizzati dal primo argomento e restituisce il controllo al chiamante NON SENZA aver prima attivato il sistema di allarme.

In pratica è come quando si va dal benzinaio e si chiede *"il pieno o venti euro"*: il serbatoio è il primo argomento, il benzinaio è *getline*, la benzina sono i *bytes*, la pompa è l'*input stream*, i venti euro sono il secondo argomento, il pieno è il terzo argomento (un po' forzata, perché dovrebbe trattarsi di una caratteristica della benzina, e NON del serbatoio...) e il *byte* terminatore è il tappo del serbatoio. E per completare la perfezione dell'allegoria, se accadesse che il serbatoio si riempia perfettamente (rilevamento del terzo argomento) nel preciso istante in cui sulla pompa scoccano i venti euro (estratto esattamente IL MASSIMO della quantità di benzina), il benzinaio metterebbe il tappo SENZA ATTIVARE ALCUN ALLARME (ossia senza dire: *"ce ne sarebbe entrata di più"*). Da ultimo l'attivazione del sistema d'allarme ha la sua controparte nel fatto che qualsiasi benzinaio umano, o pompa di benzina reale, fermerebbe l'erogazione prima di prosciugare la cisterna spargendone tutto il contenuto sul selciato della stazione di servizio...   
  
Il terzo *overload* del metodo *get*, diversamente da *getline*, NON ESTRAE dall'*input stream* il *byte* fornito come terzo argomento; per tutto il resto si comporta allo stesso modo.  
  
Il secondo *overload* di *get* estrae dall'*input stream* UN SINGOLO *BYTE*, memorizzandolo nell'argomento ricevuto (si APPREZZI il segno &), a meno che, ovviamente, il tentativo di estrazione non si risolva nel superamento della fine dell'*input stream*, con le consuete conseguenze.  
  
Tutte le funzioni *get* e *getline* fin qui discusse restituiscono, al loro completamento, l'oggetto stesso che le ha eseguite, eventualmente falsificato in caso di errore; invece il primo *overload* di *get*, non ricevendo alcun argomento, esegue sullo *stream* le STESSE operazioni svolte dal secondo *overload*, ma restituisce al chiamante il valore intero del *byte* eventualmente estratto. Tale valore può quindi essere ricevuto per assegnamento dal programma chiamante, indifferentemente in variabili dichiarate *char* o *int*.  
  
Per completezza di trattazione si cita anche l'esistenza di una funzione *getline* dichiarata e definita direttamente nel *namespace std*, e che quindi prescinde da qualsiasi oggetto gestore di *input stream*; la descrizione dei suoi argomenti, però, è prematura all'attuale livello del vostro apprendimento. Data la sua evidente non essenzialità, sarà, almeno per ora, passata sotto silenzio.  
  
Il seguente breve programma illustra il comportamento di *getline*: eseguitelo due volte, digitando, ad esempio, la prima volta *topolino*, o anche qualcosa di più corto, magari con uno spazio in mezzo, come *fa sol*, e la seconda volta *topolinia*...sempre terminando, ovviamente, con la pressione del tasto Invio... è LUI che immette nell'*input stream standard* il *byte '\n'*.

# include <iostream>  
using namespace std;  
  
int main(  )  
{   
char \*p = new char[20];  
cin . getline(p, 9);  
cout << p << '\n';  
if(!cin) cout << "stavi per far uscire la benzina...\n";  
}

Tre metodi molto interessanti sono *peek*, *ignore* ed *eof*: nessuno dei tre nasce per MEMORIZZARE NEL PROGRAMMA *bytes* provenienti dall'*input stream*, ma le azioni che compiono sono ugualmente utilissime allo *scriptor callidus*, la specie positivamente evoluta dallo *scriptor inops*, comunemente chiamata *programmatore astuto*.  
  
Il metodo *peek* non riceve alcun argomento e restituisce il valore del *byte* pronto per essere letto, ossia quello che si trova nella posizione corrispondente al valore attuale dell'indice di lettura, SENZA ESTRARLO dallo *stream* e lasciando quindi INTATTO il valore dell'indice.  
  
Il metodo *ignore*, invece, riceve DUE argomenti, entrambi *standard*, il che significa che la sua esecuzione può essere invocata in tre modi diversi, trasferendogli rispettivamente due argomenti, uno solo (il primo), o nessuno.  
  
Come manifesta il suo stesso nome, serve a ignorare *bytes* provenienti dall'*input stream*, vale a dire che li oltrepassa, aggiornando per ciò stesso il valore dell'indice di lettura, e li butta nel cesso, non avendo ricevuto ALCUN indirizzo di memoria in cui depositarli come invece ricevono *get*, quando riceve qualcosa, o *getline*.  
  
I criteri operativi di *ignore* sono dati dai valori degli argomenti che riceve: il primo è un intero, di valore *standard* pari a 1, che serve a dire QUANTI *BYTES* ignorare **AL MASSIMO**, a partire, ovviamente, dalla posizione attuale nello *stream*. Pertanto, l'invocazione di *ignore* trasmettendogli UN parametro (sia *n*) o nessuno equivale a invocare rispettivamente *seekg(n, ios :: cur)* o *seekg(1, ios :: cur)*, compreso il trattamento dell'eventuale superamento della fine dello *stream*, pagando il prezzo dell'estrazione dei *bytes* che *seekg* NON paga; e quindi non conviene usare *ignore* al posto di *seekg*....  
  
Quando però si fornisca anche il secondo argomento, che ha come valore *standard* il semplice rilevamento della fine dello *stream*, con annessa e coerente segnalazione, il prezzo pagato per l'estrazione vale la candela, perché il secondo argomento, di tipo *char*, indica a *ignore* di fermarsi e smetterla di ignorare *bytes* QUANDO SI ACCORGESSE DI AVER APPENA BUTTATO NEL CESSO un *byte* di valore UGUALE a quello ricevuto come secondo argomento: si tratta quindi di un *seekg* intelligente che non si limita a pagaiare scriteriatamente lungo lo *stream*, ricoprendo una distanza fissa assegnata *a priori*, ma pagaia guardandosi attorno, fino a quando trova un luogo interessante, interessante per il programmatore, beninteso, dove fermarsi. Naturalmente, se quel luogo interessante NON si trova entro il numero di *bytes* previsti dal primo argomento, la funzione si arresta comunque dopo aver pagaiato per QUEL numero di *bytes*, e sempre che non finisca lo *stream* prima ancora....  
  
Infine il metodo *eof*, che non riceve argomenti, restituisce un valore booleano pari sempre a *false*, TRANNE NEL CASO IN CUI **LA PIÙ RECENTE OPERAZIONE** compiuta sullo *stream*, e capace di modificare il valore dell'indice di lettura, LO ABBIA PORTATO A SUPERARE LA FINE dello *stream* medesimo.  
  
Sfruttando ciascuno di questi tre metodi, il seguente programma di sole 42 righe, la maggior parte delle quali puramente folkloristiche, è capace di ricopiare su terminale TUTTE e SOLE le linee di un documento testuale, di lunghezza IGNOTA, in numero IGNOTO e senza alcun ordinamento NOTO, che abbiano una determinata lettera INIZIALE: pensate a un documento che contenga, uno per riga, i nomi dei vostri innumerevoli morosi/e, scritti alla rinfusa, ma a partire comunque dall'inizio della riga solo per semplificarsi un po' la vita; ricordatevi comunque che TUTTO è possibile a uno *scriptor callidus*.  
Preparatevi un siffatto documento, anche se avete un/a solo/a moroso/a o perfino nessuno/a, ed **ESEGUITE IL PROGRAMMA**.

# include <iostream>  
# include <fstream>  
using namespace std;  
  
int main(  )  
{  
bool b;  
int conta = 0;  
const char \* PapioPapio = "chiama il \"Papio papio\" e lascia fare a lui\n";  
char \* nome\_del\_file = new char[201], c, \*s;  
cout << "dimmi il nome del documento\n(200 caratteri al massimo E SENZA SPAZI!) ",  
cin >> nome\_del\_file;  
do  
{c = cin . get(  );  
b = c != ' ' && c != '\t' && c != '\n'; if(b) break;}  
while(c != '\n');  
if(b)  
{cout << "avevo detto SENZA SPAZI; "<< PapioPapio; return 1;}  
ifstream is(nome\_del\_file);  
if(!is)  
{cout << "non si riesce a trovare " << nome\_del\_file << "; " << PapioPapio;  
return 2;}  
cout << "quale iniziale vuoi ? ", cin >> c;  
if(c < 'A' || c > 'Z' && c < 'a' || c > 'z')  
{cout << "Analfabeta! Non mi hai dato una lettera! " << PapioPapio;  
return 3;}  
while(true) // il numero di linee contenute è ignoto  
{while(is . peek(  ) != c)  
{is . ignore(20000, '\n'); if(b = is . eof(  )) break;}  
if(b) break;  
streamsize qui = is . tellg(  );  
is . ignore(20000, '\n'); // la lunghezza delle linee è ignota  
streamsize qua = is . tellg(  );  
is . seekg(qui),  
s = new char[qua-qui+2], // attenzione a questa linea  
is . getline(s, qua-qui+2, '\n'),  
++ conta,  
cout << s << '\n';}  
is . clear(  ),  
is . close(  );  
if(!conta) cout << "non havvi alcuno/a\n";  
}

Il programma ha almeno una superfluità e DUE CREPE evidentissime: la superfluità consiste nell'esecuzione dell'accoppiata *is . clear(  ), is . close(  );* quando il programma sta comunque finendo. Non si tratta però di una cattiva abitudine, tutt'altro....  
  
Quanto alle crepe, **provate a pensarci un attimo da soli prima di proseguire la lettura**, e se ne trovate di diverse da quelle che vi segnalerò appresso, segnalatele a vostra volta. **FATELO!** È un utile esercizio.  
  
La prima crepa è addirittura grossolana e consiste nel fatto di confrontare con le iniziali che sono scritte nel documento SOLO il carattere digitato da tastiera e non anche le sue versioni minuscolizzata o, al contrario, maiuscolizzata: il risultato è che, se il documento contiene sia una linea *Alessandra* sia una linea *alessia*, O l'una O l'altra sarà comunque omessa nell'*output* (**trovate come rimediarla**).  
  
La seconda crepa è più sottile e riguarda una **cattiva gestione della memoria**, come avrebbe dovuto suggerirvi il commento *"attenzione a questa linea"*; in effetti, a ogni passaggio nel ciclo *while*, viene presa nuova memoria, senza aver rilasciato quella del giro precedente...  
  
Anche la costante esplicita 20000 trasmessa al metodo *ignore* non è troppo elegante, ma si presume, con giusta ragione, che il nome di un moroso/a debba essere di qualche ordine di grandezza più breve, sicché il carattere *'\n'* venga incontrato BEN PRIMA di aver digerito addirittura 20 *kilobytes*.  
  
Notate piuttosto come l'uso della variabile booleana *bool b;* consenta di interrompere contemporaneamete ENTRAMBI i cicli *while*, come è NECESSARIO che accada.  
  
Apprezzate anche come, attraverso il ciclo *do*, il programma riesca ad accorgersi se si tenta di truffarlo digitando nomi di documenti con spazi IN MEZZO: non in testa, che sono comunque IGNORATI dall'estrattore, ricordàtevelo, né in fondo, che sarebbero comunque irrilevanti perché IGNORABILI a una successiva lettura.   
  
Il metodo di lettura **binaria** più **utile** e più semplice e versatile per ricavare da un *input stream* NON FORMATTATO valori per dati numerici, non importa se interi o non interi, è senza alcun dubbio il metodo *read*, il cui uso è altissimamente raccomandabile, **specialmente in ambiente scientifico**.  
  
Questo metodo riceve DUE argomenti: il primo è un puntatore a *char*, adeguatamente inizializzato, al cui indirizzo saranno ascritti i *bytes* estratti dallo *stream*, e il secondo è una variabile di tipo *streamsize* il cui valore indica al metodo QUANTI *bytes* estrarre.  
Naturalmente, se prima che l'estrazione sia completata si incoccia la FINE dello *stream*, vengono attivate come d'abitudine tutte le opportune segnalazioni d'errore e la memorizzazione dei dati **non può essere considerata andata a buon fine**.  
  
Si sottolinea che la lettura binaria di dati numerici è di gran lunga preferibile rispetto alla lettura di *stream* testuali per almeno tre ragioni:

1. non soffre di **alcuna perdita di precisione** quando si tratti di acquisire dati numerici di tipo non intero;
2. è molto più veloce rispetto alla lettura testuale, dato che non abbisogna di alcuna conversione alla rappresentazione binaria interna (e se i dati da leggere sono tanti...);
3. i documenti che contengono dati binari possono occupare sul disco, a parità di informazione contenuta, fino ad assai meno della metà dello spazio occupato da un documento testuale.

D'altro canto esistono DUE sole controindicazioni al mantenimento di informazione numerica di pregio sotto forma di documenti binari, una irrilevante e l'altra che può cagionare qualche problema, se non se ne è consapevoli:

1. la controindicazione irrilevante è che un tal genere di documenti non è comprensibile all'occhio umano: ma allora per che cosa esistono i calcolatori e i linguaggi di programmazione?
2. l'altra controindicazione consiste nel fatto che, esattamente a causa del punto precedente, tali documenti devono essere stati scritti PER FORZA proprio da un calcolatore, e si vedrà come nel canto sull'*output*, e quindi la struttura fine del loro contenuto DIPENDE dalla macchina che li ha scritti: finché si rimane sulla STESSA macchina, o su macchine di architettura UGUALE o COMPATIBILE, non si ha alcun problema; ma se un tale documento venisse fatto leggere a un calcolatore di architettura DIVERSA o comunque NON COMPATIBILE, la lettura non potrebbe essere fatta bovinamente, ossia con lo stesso codice con cui si sarebbe fatta entro il proprio orticello, ma occorrerebbe una reinterpretazione appropriata dei *bytes* estratti. Fortunatamente le architetture DIVERSE esistenti sono solo 2 [DUE].

Il seguente programma mostra come si possa leggere il valore di una variabile intera da uno *stream* binario; si utilizzerà come supporto dello *stream* un documento di QUATTRO *bytes* denominato, nell'esempio, *pappo*, in cui siano stati scritti, nell'ordine, il carattere spazio e tre caratteri nulli, in modo da poter constatare, per ispezione diretta del documento, l'incomprensibilità del contenuto dello *stream* da parte di un occhio umano (...NON ALLENATO...). Come al solito siete invitati a FAR ESEGUIRE IL PROGRAMMA, dopo aver imparato, nel prossimo canto, come preparare un documento che abbia il contenuto proposto. Per carità: CHE NON VI VENGA IN MENTE di scrivere col *text editor* uno spazio e tre zeri: non è questo...:

# include <iostream>  
# include <fstream>  
using namespace std;  
  
int main(  )  
{  
int i;  
ifstream is("pappo");  
is . read((char \*)&i, sizeof(int));  
cout  
<< "l'intero letto dovrebbe essere trentadue ... : "  
<< i   
<< "\n(chi l'avrebbe detto?)\n";  
if(is) cout << "stream in piena efficienza\n";  
}

Si apprezzi come è stata condita la variabile *i* per renderla gradita a *read* come proprio primo argomento; e anche il fatto che, a lettura conclusa, lo *stream* sia rimasto vivo e vegeto.  
  
La vera forza del metodo *read* è comunque quella di minimizzare il numero di accessi al disco fisso per compiere operazioni di lettura, accessi che sono dispendiosissimi, sia in termini di prestazioni del programma sia di usura del disco stesso: in questo senso va sempre preferita, ogni volta che sia possibile, la lettura di interi banchi di dati da memorizzare in *array* o puntatori piuttosto che di singole variabili come nell'esempio appena mostrato. Per intendersi, il seguente segmento di programma, che legge un milione di interi da un documento su disco (si suppone che CI SIANO)

ifstream is("pappo");  
int \*dati = new int[1000000];  
for(int i=0; i < 1000000; ++i)  
is . read(reinterpret\_cast<char \*>(dati+i), sizeof(int));

quantunque sia perfettamente legittimo per il compilatore, in realtà **è una schifosa orrendezza** rispetto a quest'altro:

ifstream is("pappo");  
int \*dati = new int[1000000];  
is.read(reinterpret\_cast<char \*>(dati), 1000000\*sizeof(int));

che effettua UN SOLO ACCESSO su disco piuttosto che UN MILIONE DI ACCESSI... la parola di vocabolario *reinterpret\_cast* è già stata incontrata nel canto sugli operatori e, con la presente sintassi, svolge la stessa operazione di *casting* compiuta dall'operatore *(char \*)*.  
  
È perfino consigliabile, ogni volta che la capacità di memoria del calcolatore lo permetta, effettuare COMUNQUE UN SOLO ACCESSO, o il MINIMO numero di accessi, sul disco fisso, anche quando i dati da leggere fossero del tutto eterogenei, e quindi tali da non poter essere indirizzati tutti da un unico puntatore, e demandare le effettive operazioni di lettura a un oggetto gestore di un *input stream* in memoria, incomparabilmente più veloce ed efficiente: ecco perché esistono gli *stream* con supporto in memoria. Per esemplificare quanto asserito, eccovi un appropriato segmento di codice per la lettura di un documento binario contenente sia interi sia non interi, in alternanza; ESEGUITELO e domandatevi il significato della linea che recita *dati[n-1]=0;*:

int \*dati\_interi = new int[1000], n;  
double \* dati\_non\_interi = new double[1000];  
char \* dati =   
new char[n=1+1000\*(sizeof(int)+sizeof(double))];  
ifstream stream("pappo");  
stream.read(dati, n-1); // UN SOLO ACCESSO  
dati[n-1] = 0;  
istringstream issa(string(dati,n));  
for(int i=0; i < 1000; ++i) // questo sì!  
issa.read(reinterpret\_cast<char\*>(dati\_interi+i), sizeof(int)),  
issa.read(reinterpret\_cast<char\*>(dati\_non\_interi+i), sizeof(double));

Osservate l'inizializzazione dell'oggetto *issa*, che avviene per il tramite di un altro oggetto di tipo *string* contenente esattamente *n* caratteri, per usare il quale occorre un'inclusione appropriata: *#include <string>*. Si tratta di una scelta NECESSARIA perché se nel documento binario vi fosse, come è chiaramente possibile, qualche *byte* di valore nullo PRIMA DELLA FINE del documento, tale *byte* nullo costituirebbe la FINE dell'oggetto *stream* governato da *issa* se la sua inizializzazione fosse stata compiuta direttamente tramite il puntatore *dati*, come si era fatto finora. Ed era questo il significato da dare a quello che vi avevo raccomandato di domandare a voi stessi.  
  
Imparerete a tempo debito che cosa siano gli oggetti di tipo *string*...   
  
  
Ci sono ancora pochi altri metodi di cui si ritiene opportuno parlare, oltre a quelli che sono stati, o saranno, volutamente taciuti (*readsome*, *rdbuf*, *sync*, *swap*, *imbue*, *tie*, *narrow*, *widen*, *init*, *move*...tutti i metodi degli aggregati *streambuf*, *stringbuf*, *filebuf* e *sentry*... solo per darvi un'idea di quanto sia enorme la parte sommersa dell'*iceberg*...): si tratta di *gcount*, *unget* e *putback*.   
  
Il primo (*gcount*) non riceve alcun argomento e restituisce al chiamante il numero di *bytes* estratti dallo *stream* in seguito alla **più recente** operazione di lettura binaria, indipendentemente dal fatto che sia andata a buon fine o no; attenzione: NON i *bytes* estratti dall'estrattore, che NON effettua lettura binaria. In pratica i metodi che aggiornano il valore potenzialmente restituito da *gcount* sono *read*, *getline* e *get*, in ogni suo *overload*; invece *seekg*, per esempio, lo lascia intatto. Ecco un esempio d'uso, che illustra quanto appena detto (**ESEGUITELO!**):

# include <iostream>  
# include <sstream>  
using namespace std;  
  
int main(  )  
{  
istringstream is(" 10 20 30.6 ");  
int i;  
char c[  ] {0,0,0,0};  
is >> i,  
cout  
<< "letto " << i   
<< " per complessivi "   
<< is . gcount(  )   
<< " byte\n";  
is . seekg(0),  
is . read(c, 3);  
cout   
<< "letto \b"   
<< c << " per complessivi "  
<< is . gcount(  )  
<< " byte\n";  
is . seekg(0),  
cout  
<< "seekg non altera gcount: "  
<< is . gcount(  ) << '\n';  
c[0] = is . get(  ),  
cout  
<< "letto ["  
<< c[0]  
<< "] per complessivi "  
<< is . gcount(  )  
<< " byte\n";  
}

Nel precedente programma *\b* NON È un errore; si osservi che cosa accade se si facessero leggere a *read* 300 *bytes* invece di 3...se il programma proseguisse, verosimilmente andrebbe incontro a errore per **violazione di memoria indirizzata da *c***, non per colpa di errori di lettura...  
  
I metodi *unget* e *putback* sono simili, ma non uguali, e non solo per il fatto che il primo non riceve alcun argomento, mentre il secondo ne riceve UNO, di tipo *char*.  
  
I manuali, di solito, citano *unget* come il metodo che *"rende novamente disponibile nello stream l'ultimo byte estratto"*, adombrando che sia una sorta di negazione di *get*, come suggerirebbe il suo stesso nome.  
Questa definizione, tuttavia, non trova d'accordo l'autore di queste note, e per i seguenti motivi:

1. il metodo NON riceve argomenti. Se la definizione citata fosse rigorosamente corretta, ne dovrebbe essere previsto almeno un *overload* tale da dare il significato di *"nessuna operazione sull'oggetto stream"* a un'espressione come *stream.unget(stream.get(  ))*, il che NON avviene;
2. la frase *"rende novamente disponibile"* è, in sé, un'aberrazione: i *bytes* di un *input stream* che abbia supporto su disco, tanto per dire, non cessano MAI di essere disponibili, perché il supporto di un *input stream* non subisce MAI modifiche o cancellazioni di sorta: ci mancherebbe altro!
3. se poi la frase *"rende novamente disponibile"* volesse solo sottintendere che il prossimo *byte* a essere estratto dallo *stream*, subito dopo l'azione di *unget*, sarà novamente lo stesso *byte* estratto per ultimo, subito prima dell'azione di *unget*, neppure questo è vero, come dimostra con evidenza il seguente codice (**ESEGUITELO!**).

# include <iostream>  
# include <sstream>  
using namespace std;  
  
int main(  )  
{  
istringstream is(" 10 20 30.6 ");  
cout  
<< "all'inizio sono in "  
<< is . tellg(  ) << '\n';  
char c = is.get(  );  
cout  
<< "dopo get sono in "  
<< is . tellg(  ) << " con "  
<< is . gcount(  )   
<< " con valore [" << c << "]\n";  
is.seekg(2, ios::cur),  
cout  
<< "dopo seekg sono in "   
<< is . tellg(  ) << " con "  
<< is . gcount(  ) << '\n';  
is . unget(  );  
cout  
<< "dopo unget sono in "  
<< is . tellg(  ) << " con "  
<< is . gcount(  )  
<< " e se leggo, trovo [";  
cout << (c = is.get(  )) << "]\n";  
}

In sostanza, secondo l'autore, e anche secondo il modo di agire di *unget* testé dimostrato, la definizione corretta è che questo metodo semplicemente retrocede di un *byte*, quando ciò è possibile, la posizione dell'indice di lettura, ma nondimeno NON equivale a *seekg(-1, ios::cur)* perché quest'ultima NON azzererebbe il valore restituito da *gcount*, come invece *unget* fa.  
  
Quanto al metodo *putback*, essendo il solo, fra quelli che vi saranno presentati, che, pur appartenendo all'aggregato degli oggetti gestori di *input stream*, ha anche la potenzialità di modificarne il contenuto, in determinate situazioni, se ne discuterà solo dopo aver contemplato adeguatamente gli *stream* bidirezionali.  
  
Per concludere questo corposo e faticoso canto, resta solo da affrontare il tema della riparazione degli errori in cui un programma può incorrere durante le operazioni di INPUT, dovuti sia a imperizia del fruitore del codice sia a oggettive e inevitabili situazioni di difficoltà che possono verificarsi a detrimento dell'*input stream*.  
  
Al fine di una corretta e liberatoria gestione di tali situazioni, il sistema *ios* prevede che, in ognuno degli aggregati costituenti gli oggetti deputati a interfacciare il programma con l'*input stream*, sia collocata una speciale variabile il cui valore corrente ne rispecchia il cosiddetto *stato*: quando lo *stream* gode di ottima salute il valore di quella variabile è 0 (zero); in caso contrario la variabile assume automaticamente un valore NON NULLO diverso, secondo il tipo di errore verificatosi.  
  
Il metodo *rdstate(  )*, che non riceve argomenti, restituisce al chiamante il valore corrente di tale variabile, di tipo *std :: ios\_base :: iostate*, un tipo riconducibile a qualche modificazione di *int* dipendente dall'architettura della macchina ospite, mentre il metodo *clear(std :: ios\_base :: iostate = 0)*, che riceve l'argomento *standard* indicato, le assegna il valore trasmesso; questo metodo è stato già incontrato, eseguito senza argomenti, con l'intento appunto di disattivare il sistema di allarme; capite adesso che cosa significasse quella frase.  
  
Esiste, per la verità, anche un metodo *setstate* che riceve UN argomento obbligato, vale a dire NON *standard*, dello stesso tipo di quello ricevuto da *clear* e che rispetto a quest'ultima agisce in maniera cumulativa, aggiungendo il valore trasmesso al valore corrente, piuttosto che sostituirlo come fa *clear*, ma se ne può intuire, per questa stessa ragione, la rara utilità. "Aggiungere" ha qui un significato che sarà chiaro entro poche righe.  
  
Ovviamente il risolutore di ambito del *namespace std*, ossia *std::*, può essere omesso, ovunque appaia, se si è inserita nel codice la linea *using namespace std;*.   
  
Qualcuno forse dirà, e se non lo dice significa che non sta capendo un tubo: SVEGLIAAAAA!: *"e come faccio a sapere quali sono i possibili diversi valori non nulli di quella variabile e quale sia il loro significato? Una volta che ne avessi ottenuto il valore eseguendo rdstate(  ), che me ne faccio, se non è zero?"*...  
  
Domanda legittima, che ha come risposta *"nessuna preoccupazione!"*  
  
In effetti sono previste, come al solito, delle costanti dichiarate negli aggregati *std::ios\_base* il cui solo nome ne spiega il significato e il cui valore costante è quello che la variabile di cui si parla assume, nelle diverse circostanze corrispondenti al nome che quelle costanti hanno.  
Non ha, a questo punto, alcuna importanza il loro valore numerico esplicito, che perfino potrebbe essere DIVERSO da un compilatore a un altro quantunque il valore 0 (zero) sia mantenuto praticamente da tutti come valore denotante la condizione di NON ERRORE; basta solo che il risultato prodotto da *rdstate(  )* sia confrontato con queste costanti dichiarate, citandole per nome anziché per valore: capite perché un programma fa bene a non contenere costanti esplicite?  
  
Ecco i nomi delle costanti dichiarate (sono QUATTRO), che vanno SEMPRE prefissati col risolutore *ios\_base ::* , o con la sua abbreviazione *ios ::* , e FORSE ANCHE, E PRIMA, col risolutore opzionale *std ::*

* *goodbit*
* *badbit*
* *eofbit*
* *failbit*

Si tratta o no di nomi parlanti? Per inciso il suffisso comune *bit* lascia capire che i loro valori non nulli, quelli di *badbit*, *eofbit* e *failbit*, sono quelli di potenze di 2: quali potenze non ha alcuna rilevanza, tanto il valore finale della variabile restituita da *rdstate(  )* è il risultato dell'azione dell'operatore binario *|=* compiuta ricorsivamente aggiungendo (ricordate quel che faceva *setstate*, poche righe fa?), secondo le contingenze, una delle costanti, come operando destro, al valore attuale contenuto nell'operando sinistro.  
  
Ora dovreste capire anche il motivo per cui tutti i compilatori convengono che il valore di *goodbit* sia zero, valore neutro per l'operatore *|=*, essendo la costante che indica lo stato dello *stream* sgombro di errori (*good*, appunto).  
  
Appena si incorre in una condizione di errore qualsiasi, il metodo che vi incappa ESEGUE AUTOMATICAMENTE, **PER VOI**,  
  
           *setstate(rdstate(  )  |  ios::.....)*   
  
mettendo, al posto dei puntini, la dovuta e opportuna costante o una cumulazione di più di una sola, in modo tale che, alla successiva esecuzione di *rdstate(  )*, otteniate il valore aggiornato della variabile di *stato* interna. È chiaro, no?  
  
Siccome può poi essere noioso controllare quali siano i suoi *bit* accesi, per poter prendere le decisioni del caso, ossia scrivere cose come questa, pure legittima ed efficace:  
  
*if(cin . rdstate(  ) == ios :: eofbit) cout << "facciamo qualcosa!\n";*   
  
ci pensa il sistema stesso, provvedendo delle funzioni di valore booleano e senza argomenti che permettono di sostituire la precedente con questa, **DEL TUTTO EQUIVALENTE**:   
  
*if(cin . eof(  )) cout << "facciamo qualcosa!\n";*   
  
ove ricompare il metodo *eof(  )* che si era già incontrato e che ora si riconosce restituire null'altro, per la pignola esattezza che contraddistingue l'autore, che il valore booleano di *(rdstate(  ) & ios :: eofbit) == ios :: eofbit*.  
  
Naturalmente, al posto dell'emissione di un messaggio degno solo dei tre avvoltoi del disneyano *il libro della giungla*, va fatto effettivamente qualcosa: trattandosi, in questo caso, del superamento della FINE dello *stream* che, se ripercorrete questo stesso canto, si è visto possa accadere in una DOVIZIA DI OCCASIONI, non trattandosi neppure di un vero e proprio errore, in definitiva, ma di una circostanza del tutto naturale e prevedibile, la cosa da poter fare potrebbe essere una delle seguenti:

1. terminare il programma: QUESTA potrebbe anche essere la cosa giusta;
2. uscire da un ciclo eterno, indi far eseguire *clear(  )* ed, eventualmente, *close(  )* dall'oggetto gestore dello *stream* in vista di un suo potenziale riutilizzo, e proseguire l'esecuzione facendo tesoro di quanto è stato letto: QUESTA è quella che dovreste scegliere nella schiacciante maggioranza dei casi;
3. se si fosse consci di trovarsi nell'ambito di *try* e si sapesse che esiste qualche *catch* predisposto a gestire in modo ottimale la sopravvenuta condizione, eseguire l'operatore *throw* con un operando adeguato: anche QUESTA non sarebbe male, ma, per forza di cose, si può scegliere più raramente ed è efficace *"solo se..."*
4. raccomandarsi alla buona sorte: QUESTA è degna di uno *scriptor inops*;
5. correre dal vostro docente: QUESTA la sceglierete in tanti...

È abbastanza scontato che, accanto al metodo *eof(  )*, ci siano, con significato perfettamente analogo, anche *bad(  )* e *fail(  )* e perfino un metodo *good(  )*, anche se, per quest'ultimo, il valore restituito è semplicemente quello di *rdstate(  ) == ios :: goodbit* o magari quello di *!rdstate(  )*, potendo ognuno essere sfruttato per fare, all'occorrenza, la cosa giusta; e in effetti CI SONO.  
  
Resta solo da capire in quali circostanze si accendono i *bit* relativi a *badbit* e *failbit*, ritenendo di essersi ormai diffusi abbastanza sulle altre due costanti.  
  
Come dice la parola stessa *badbit* è proprio meglio che non si accenda, perché se si accende è segno *bad*: lo *standard* del linguaggio recita che l'accensione di *badbit* si verifica in seguito a (citazione): *"non-recoverable errors"*, quindi se, per definizione, l'errore è irrecuperabile, vuol dire che non c'è niente da fare. Riferendosi all'elenco precedente si può solo ricorrere al punto 1., se si fa a tempo...probabilmente il disco si sta rompendo...a meno che lo *scriptor inops* non stia spargendo in giro per il codice dei puntatori nulli con la pretesa di utilizzarli...  
  
L'accensione di *failbit* è qualcosa di meno grave e riguarda essenzialmente il comportamento dell'*alouatta belzebul* (scimmia urlatrice dalle mani rosse) al calcolatore: si verifica infatti quando non si riesce ad aprire uno *stream* per aver fornito un nome sbagliato per il documento di supporto, oppure quando si pretende di far leggere un intero all'estrattore e si digita, al posto di numeri, il proprio codice fiscale. In questi casi, se ricordate, e specialmente in caso di lettura formattata, lo *stream* **va ripulito**, altrimenti dalla situazione di errore NON SI ESCE. Riconsiderate in proposito il programma proposto in questo stesso canto allorché si chiedeva un nome che non contenesse spazi.  
  
Anche il seguente segmento di codice illustra come fronteggiare una situazione di *failbit* acceso

int n;  
do {   
cout << "digita un intero\n";// ci si aspetta il peggio  
cin >> n;  
// se l'*alouatta* digita, a questo punto,  
// fadkladfkv489fgu9  
// ....  
if(cin . fail(  )) // *failbit* è acceso, credetemi  
{  
char c; // confinato in questo ambito  
cin . clear(  ); // primo: spegnere *failbit*  
do ; // "digerisci" il ciarpame  
while((c = cin . get(  )) != '\n');  
// fino a quando non trovi "Invio"  
// e adesso che tutto è pulito,  
// visto che siamo in un ciclo,  
// fai ridigitare  
continue;  
}  
// hai risposto bene, scimmietta  
break;  
} while(true); // fino a quando non hai risposto giusto

Ovviamente situazioni di lettura più complesse richiederanno maggior attenzione, ma il canovaccio resta lo stesso. Non bisogna poi dimenticare che, quando la variabile di stato ha un valore diverso da *goodbit*, l'oggetto stesso, ad esempio *cin*, risulta booleanamente falsificato, nel senso che *!cin* è un'espressione booleana di valore *true* e, **paradossalmente** *cin* è un'espressione booleana di valore *false*, in apparente contraddizione con l'assioma *TUTTO È VERO QUEL CHE NON È ZERO*.

**Quarta pausa di riflessione**

Dopo aver riletto con attenzione tutti i canti fin qui proposti, dal canto 0 al canto 20, e specialmente DOPO ESSERVI SCIROPPATI il canto sull'INPUT appena trascorso, meritate una mezza giornata di vacanza.  
Se vi va di provare lo stato attuale della vostra SCIENZA, scrivete un programma che si destreggi nella lettura di documenti di ogni genere presenti nelle vostre cartelle, tenendo presente la differenza che passa fra quei documenti che riuscite a capire anche voi e quelli che non riuscite a capire affatto: questi ultimi sono quelli BINARI.  
Potreste, ad esempio, scriverne uno che, leggendo il SUO STESSO ESEGUIBILE (*a.out*), vi scriva sul terminale la statistica del numero di occorrenze di ciascuno dei 256 *bytes* ASCII al suo interno.   
  
In ogni caso, quando tornerete a leggere, tornate senza dimenticare spento alcun neurone, perché avrete bisogno del contributo di TUTTI QUELLI che avete.

**Canto ventunesimo: il sistema *ios* per l'OUTPUT**

Questo canto sarà più riposante rispetto al precedente, sia perché ormai si dovrebbe essere adeguatamente allenati sia perché ci sono oggettivamente meno cose da dire.  
  
L'unica maggiore complicazione dell'OUTPUT rispetto all'INPUT riguarda proprio gli oggetti precostituiti: mentre nell'INPUT ce n'era uno solo (*cin*), nell'OUTPUT, oltre a *cout*, che si conosce da quando è nato il mondo, e che si è SEMPRE UTILIZZATO, fin dall'alba del mondo, ce ne sono ben altri DUE, denominati *cerr* e *clog*; per il resto, come si era anticipato, l'OUTPUT è costituzionalmente meno intricato dell'INPUT, se non altro perché il programma ha già nella propria memoria quel che deve finire nello *stream*.   
  
La ragione dell'esistenza di TRE oggetti invece di UNO risiede nel fatto che i sistemi operativi della famiglia UNIX, a fronte di UN SOLO *standard input stream* previsto, prevedono, accanto al corrispondente *standard output stream*, gestito dal C++ tramite *cout*, un cosiddetto *standard error stream*, gestito in cooperazione da *cerr* e *clog*; il supporto fisico di ENTRAMBI gli stream è, salvo avviso contrario, SEMPRE il terminale da cui il programma viene eseguito, di modo che OGNI COSA che viene scritta da ciascuno dei tre oggetti finirebbe, in assenza di contrordini, mischiato, in modo indistinto, sul terminale.  
  
L'aggettivo indistinto non è scelto a caso, come NULLA di quel che trovate in queste note è scritto a caso, dato che, in un contesto di programma cosiddetto *multithreaded*, il cui significato non è ancora tempo di spiegarvi, potrebbe perfino succedervi di vedere scrivere PRIMA quello che avreste, erroneamente, pensato di avere fatto scrivere DOPO.  
  
Per evitare di trovarsi in imbarazzi di questo tipo ci sono numerosi sistemi, che vanno dal continuare a usare SOLO *cout*, come si è fatto finora, a un utilizzo pienamente consapevole e INTELLIGENTE dei due *stream* e dei tre oggetti. Il fatto poi che uno *stream* sia gestito da DUE oggetti distinti è implicato da alcune ulteriori sottigliezze che non mette conto approfondire per il vostro livello di apprendimento: eventualmente domandatelo lungo i corridoi; se vi interessa un consiglio usate *cout* per il VERO OUTPUT del programma, ossia per i risultati che volete ottenere, usate *clog* per far uscire messaggi provvisorii, con eventuali annessi valori di variabili, in fase di allestimento del codice, così da poterli trovare facilmente con il pulsante di ricerca del vostro *text editor* quando voleste eliminarli, e usate *cerr* per far uscire messaggi di controllo durante la caccia al baco, sport che vi capiterà spessissimo di dover praticare.  
  
In questo modo, giovandovi anche delle proprietà del sistema Linux a proposito del reindirizzamento degli *stream standard*, si possono separare in qualsiasi momento i due *output stream* e anche abolire completamente uno qualsiasi dei due, senza neppure dover ricompilare il codice. Quanto appena detto può essere efficacemente esemplificato dal seguente codice elementare

# include <iostream>  
using namespace std;  
  
int main(  )  
{  
cout << "sono cout\n";  
clog << "sono clog\n";  
cerr << "sono cerr\n";  
}

eseguendolo in successione, senza mai ricompilarlo, e IN UNA CARTELLA **VUOTA**, nei seguenti modi:

1. ./a.out
2. ./a.out   >  *file1*
3. ./a.out   >  */dev/null*
4. ./a.out   2>  *file2*
5. ./a.out   2>  */dev/null*
6. ./a.out   1>  *file1\_1*    2>  *file2\_1*
7. ./a.out   1>  *file1\_3*    2>  */dev/null*
8. ./a.out   1>  */dev/null*    2>  *file2\_4*
9. ./a.out   1>  */dev/null*    2>  */dev/null*
10. ./a.out   &>  */dev/null*
11. ./a.out   &>  *file12\_5*

Al termine di ogni esecuzione prestate attenzione a quel che appare sul terminale e a quali documenti sono stati appena creati nella cartella, nonché al loro contenuto: scoprirete agevolmente che tutte le esecuzioni dalla numero 2. alla numero 8. comprese tengono separati i due *stream*, inviandone uno su terminale e l'altro su un documento (esecuzioni da 2. a 5.) oppure ciascuno su un documento differente (esecuzioni da 6. a 8.); scoprirete peraltro subito che il documento chiamato */dev/null* è una sorta di buco nero, dal quale NULLA è più reperibile e che, quindi, inviarvi dentro uno *stream* equivale a sopprimerlo: pensate quanto questo sia utile quando avrete trovato il baco e sarete curiosi di vedere il risultato agognato, senza doverlo CERCARE nella marea di messaggi che vi sarete fatti scrivere da *cerr*.  
  
Nelle esecuzioni 1., 9., 10. e 11. i due *stream* sono lasciati sullo stesso supporto: il terminale nella 1., il buco nero nelle 9. e 10., che sono, per ciò stesso, equivalenti, e un documento UNICO nella 11. (GRAN SISTEMA OPERATIVO).  
  
Va solo prestata attenzione a TRE cose:

* le sintassi delle undici diverse possibili esecuzioni sono quelle richieste dalla *shell bash*, che comunque È la *shell* usuale di Linux; se uno vuole usare un'altra *shell* le sintassi possono essere leggermente diverse: non si ha né tempo né voglia di mostrarle tutte;
* NON SI FRAPPONGA MAI **SPAZIO VERUNO** tra il segno > e il carattere che lo PRECEDE IMMEDIATAMENTE nelle esecuzioni dalla 4. compresa in poi, altrimenti SALTA TUTTO: il calcolatore potrebbe anche esplodervi in mano (*STO SCHERZANDO*, comunque salta tutto davvero); del resto ANCHE NELLE ESECUZIONI 2. e 3. si sarebbe potuto scrivere 1> (TUTTO ATTACCATO!) al posto del solitario, e più corto da scrivere, >;
* quando si indirizza lo *standard output stream* e/o lo *standard error stream* su un documento, CHE NON SIA */dev/null*, nei modi sopra mostrati, SE QUEL DOCUMENTO ESISTESSE GIÀ nella cartella corrente, **il suo contenuto precedente all'esecuzione VA INESORABILMENTE PERDUTO** e **SOSTITUITO** con quello che l'esecuzione di *./a.out* produce.  
  Se si volesse aggiungere quest'ultimo contenuto a quello già presente in un documento preesistente, OGNI SEGNO > andrebbe semplicemente RADDOPPIATO, sempre senza spazi frammezzo: ad esempio  
  *./a.out   >>  file1*  
  appenderebbe lo *standard output stream* generato da *a.out* in coda al documento *file1*, se questo fosse già esistente (GRANDISSIMO SISTEMA OPERATIVO).

Esaurito il discorso sugli *output stream standard*, si passerà ora a introdurre quelli su disco e su memoria, procedendo parallelamente a come si era fatto per l'INPUT: i tipi da usare sono, rispettivamente,

**ofstream  
ostringstream**

come era logico attendersi, ferme restando le stesse inclusioni effettuate per l'INPUT e l'eventuale necessità di prefissarli con **std ::**.  
  
Come si era già anticipato, tuttavia, l'apertura di uno *stream* di OUTPUT è del tutto diversa da quella di uno *stream* di INPUT: nella dichiarazione con contestuale inizializzazione,

**ofstream** output\_stream("file\_output");

il documento *file\_output* NON DEVE PER FORZA ESISTERE, anzi, delle due, è meglio che non esista affatto, perché, se esistesse, il suo contenuto finirebbe senza scampo nel rusco, esattamente come accade per il reindirizzamento dello *standard output stream* su disco usando un solo segno *>*. Occorrerà, piuttosto, che si abbia diritto di creazione di un nuovo documento, e questo solitamente accade, a meno che non si vada a eseguire il programma da una cartella non di proprietà: in tal caso, come avveniva per l'INPUT, l'oggetto *output\_stream* appena dichiarato finirebbe falsificato, nel senso già discusso, e nessuna operazione di scrittura potrebbe aver luogo tramite esso.  
  
Se ci si trovasse nel bisogno/convenienza di aggiungere dati a un documento preesistente, senza perderne il contenuto, vale a dire: fare qualcosa di analogo al reindirizzamento dello *standard output stream* usando il doppio segno *>>*, si dovrà procedere così:

**ofstream** output\_stream("file\_output", ios :: app);

o, equivalentemente:

**ofstream** output\_stream;  
output\_stream  .  open("file\_output", ios :: app);

ove *ios::app*, o anche *ios\_base::app*, è un'altra delle costanti dichiarate di cui si sono già conosciuti numerosi altri esemplari.  
  
Per quanto riguarda il tipo *ostringstream*, che serve a gestire gli *output stream* con supporto nella memoria del programma stesso, è, o dovrebbe essere, abbastanza intuibile che il metodo *str*, che pure si trova nel suo aggregato, non possa tanto servire a inizializzare lo *stream*, dato che si tratta di SCRIVERCI DENTRO, non di leggervi qualcosa già ivi contenuto, quanto a RICAVARE appunto quello che vi sia stato scritto. In effetti il metodo *str* si presenta con due *overload*, entrambi presenti tanto negli aggregati di *istringstream*, ove ne è stato presentato e discusso il primo, quanto in quelli di *ostringstream*, per i quali si presenterà il secondo, che NON riceve alcun argomento. La morale pratica è che un oggetto *ostringstream*, APPENA DICHIARATO, è già pronto a ricevere dati che il programma voglia scriverci dentro, senza bisogno di alcuna azione ulteriore.  
  
Ciò non significa che sia VIETATO inizializzare un oggetto *ostringstream*, contestualmente con la sua dichiarazione OVVERO attraverso il metodo *str*, con una stringa di caratteri, esattamente come si era fatto, di necessità, con gli *istringstream*: basta tener presente che la stringa trasmessagli sarà, salvo operazioni contrarie, SOVRASCRITTA A PARTIRE DALL'INIZIO, né più né meno di quanto accade ai documenti su disco, come già detto. A dire la verità l'analogia si spinge oltre, nel senso che se si dichiara l'oggetto *ostringstream* in questo modo, senza quindi ricorrere, stavolta, al metodo *str*:

**ostringstream** oss(" 10 20 30 ", ios :: ate);

con *ios::ate* che è ancora una volta una costante dichiarata il cui significato, neanche troppo celato, è *"at end"*, le operazioni di scrittura sullo *stream* si accoderanno appunto al contenuto della stringa inizializzante.  
  
Quando il programma avesse bisogno/volontà di ricuperare il contenuto ATTUALE dell'*output stream* gestito da un oggetto *ostringstream*, dopo non si sa quante operazioni di scrittura compiute su di esso, ricorrerà appunto all'*overload* SENZA ARGOMENTI dello stesso metodo *str* e segnatamente a ciò che tale *overload* restituisce al chiamante, e a questo proposito occorre prestare ALQUANTA ATTENZIONE. Considerate il seguente codice, e i commenti che vi sono inseriti:

# include <iostream>  
# include <sstream>  
using namespace std;  
  
void funza(const char \*x, const char \*y)  
{cout   
<< "funza(const char \*, const char \*)\n",  
cout  
<< x << '\n'  
<< y << '\n';}  
  
void funza(ostringstream &x, ostringstream &y)  
{cout  
<< "funza(ostringstream&, ostringstream&)\n",  
cout  
<< x . str(  ) . data(  ) << '\n'  
<< y . str(  ) . data(  ) << '\n';}  
  
int main(  )  
{  
ostringstream  
ossA("abcde", ios\_base::ate), ossB("abcde");  
ossA << 'A'; // questo si accoda  
ossB << 'A'; // questo sovrascrive 'a'  
// per ricuperare i contenuti si invoca  
// il metodo data(  )  
// che si trova nell'oggetto restituito da str(  )  
// data(  ) restituisce un puntatore a const char  
// che va USATO direttamente, piuttosto che  
// assegnarlo a puntatori dichiarati  
// nell'ambito di main(  ) come questi:  
const char  
\* a = ossA . str(  ) . data(  ),  
\* b = ossB . str(  ) . data(  );  
// l'asserto del precedente commento  
// si riscontra nel diverso output  
// generato da queste due funzioni in overload  
// (che in teoria dovrebbe essere identico)  
funza(ossA, ossB);  
funza(a, b);  
}

Se eseguirete il programma (**ESEGUITELO!**) troverete il seguente OUTPUT sul terminale, almeno se si usa il compilatore GNU 4.9.2:

*funza(ostringstream&, ostringstream&)  
abcdeA  
Abcde  
funza(const char \*, const char \*)  
Abcde  
Abcde*

Dunque la funzione *funza(ostringstream&, ostringstream&)* che USA direttamente, senza assegnarli, i puntatori restituiti da *data(  )* si comporta come ci si deve attendere, diversamente da *funza(const char \*, const char \*)* che riceve dei puntatori assegnati in *main*, i cui valori puntati sono assai prossimi al ciarpame; qualche compilatore solo un pochino più schizzinoso potrebbe già, in un simile, minuscolo, programma, incorrere in errori di accesso in memoria durante l'esecuzione.  
  
La comprensione della ragione di questa apparente incongruenza, solo APPARENTE, tuttavia, è ancora prematura, al vostro attuale livello: sappiate solo che **NON SI FA**.  
  
Potreste comunque arguire che sia celata nel fatto che sia stato necessario utilizzare due volte l'operatore punto nella stessa espressione... non che sia errato... ricordatevi di questo episodio quando si parlerà di *rvalues* e di riferimenti a *rvalues*... Alla fine di questo stesso documento, come nota a pie' di pagina, vi sarà anticipata la ragione di un simile fenomeno.  
  
In fin dei conti, visto che un *output stream* in memoria ha come precipua ragion d'essere quella di preparare i contenuti da far uscire su disco senza accedervi materialmente volta per volta, il comportamento tenuto da *funza(ostringstream&, ostringstream&)* è adeguato e irreprensibile.   
  
Proseguendo nel parallelo con gli *input stream* dovrà apparire quasi scontato che ci sia un *indice di scrittura* che può essere posizionato ove si vuole e di cui si può conoscere in ogni istante il valore attuale, con la stessa, IDENTICA, SEMANTICA dei metodi *seekg* e *tellg* che sono provvidamente sostituiti dai corrispettivi ***seekp*** e ***tellp***, in cui la *p*, iniziale del verbo angloamericano *put*, prende il posto della *g* di *get*.  
  
Tuttavia, mancando negli *output stream* il concetto restrittivo di FINE dello *stream*, per cui anche il metodo *eof(  )*, pur presente, è sostanzialmente inutile, restituendo, praticamente SEMPRE, *false*, il metodo *seekp* consente di posizionare l'indice di scrittura un numero arbitrario di *bytes*, entro ragionevolezza, OLTRE la posizione dell'ultimo *byte* già scritto, vale a dire che ha perfettamente senso invocare l'esecuzione di *seekp(1000, ios::end)* ossia con un valore POSITIVO del primo argomento; tornate a controllare che ciò NON ERA CONSENTITO a *seekg*.  
  
Una tale richiesta produce semplicemente un *canyon* di mille *bytes* NULLI nello *stream*, che potranno essere eventualmente riscritti in un momento successivo o anche lasciati così come sono.  
  
Naturalmente esiste, con la stessa funzionalità, il metodo *close(  )*, SOLO per gli oggetti *ofstream*, che hanno da contrapporgli il metodo *open*, NON per gli oggetti *ostringstream* che un tale metodo non hanno.  
  
NON ESISTE, per ovvie ragioni, un corrispettivo di *ignore*: basta NON scrivere quel che non si vuole scrivere, non trovate? E neppure un corrispettivo di *getline*, di *gcount* e di *peek*; esiste invece un corrispettivo di *get*, chiamato *put*, ma con UN UNICO *overload* che riceve, come SOLO argomento, di tipo *char*, il *byte* da immettere nello *stream*. Come vedete è vero che l'OUTPUT è meno complicato dell'INPUT...   
  
Niente corrispettivi fantasiosi come ipotetici *unput* o *getback*...va bene la fantasia, ma quando è troppa, è troppa.  
  
Esiste invece, ovviamente, e con la STESSA SEMANTICA, il corrispettivo di *read* che, naturalmente, si chiama *write* e che è il metodo d'elezione per produrre documenti binari da dare poi in pasto a *read* affinché li rilegga; a questo proposito torna in evidenza l'argomento delle possibili architetture diverse fra un calcolatore e l'altro, per cui non è detto che un documento binario scritto con *write* dal calcolatore A possa essere riletto con *read*, e senza alcuna cautela, dal calcolatore B. Quali siano le cautele da porre eventualmente in essere esula dallo scopo di questa guida: chi fosse interessato lo chieda personalmente.   
  
Tutto il sistema di segnalazione degli errori è disponibile tale e quale era nel caso dell'INPUT: tuttavia negli *output stream* le condizioni di errore si verificano estremamente più di rado. Quando però si verificano sono, in genere, più gravi, nel senso che ad essere acceso è, più spesso, *badbit* rispetto a *failbit*; ciò accadrebbe, ad esempio, quando, a furia di scrivere, SI RIEMPISSE IL DISCO: una tale condizione è chiaramente irrecuperabile da parte di un povero programma privo, tutto considerato, di una propria volontà.  
  
Rimane solo da considerare l'uso dell'operatore di inserimento (*<<*) negli *stream* testuali, che, a ben guardare, è il PRIMO operatore che sia stato usato in tutta la storia di queste note e che è l'evidente corrispettivo dell'operatore di estrazione (*>>*) dagli *input stream* testuali: entrambi gli operatori, gioverà ricordarlo, sono disponibili PER TUTTI GLI OGGETTI gestori dei rispettivi *stream*, a qualunque tipo particolare appartengano, e sono in grado di operare SU TUTTI I TIPI **NATIVI** previsti dal linguaggio, con la clausola sottintesa che, quando operano su puntatori a *char*, finiscono per leggere/scrivere i *char* che questi puntano, fino al primo spazio, in lettura, o fino al primo *byte* nullo, in scrittura.  
  
Se il programmatore non è una larva umana, i casi in cui l'operatore *<<* può incorrere in errore, per il solo fatto di essere usato, sono in numero molto vicino a zero, o comunque di gran lunga inferiore rispetto a *>>*, semplicemente per mancanza di scimmie in agguato.   
  
Tuttavia per molti (fra questi non l'autore di queste note) il comportamento, diciamo così, spontaneo, di *<<* non risulta gradevole. Eseguite, ad esempio, il seguente programma semplicissimo (**ESEGUITELO!** Altrimenti non si capirà di che cosa si stia parlando!):

# include <iostream>  
# include <cmath>  
using namespace std;  
int main(  )  
{  
cout  
<< "Questo programma scrive 31 valori "  
"di comuni funzioni elementari di "  
"una variabile\nequidistribuita "  
"nell'intervallo [1, 2].\n\n"  
"x sin(x) cos(x) tan(x) "  
"log(x) exp(x) x^0.5\n\n";  
for(int i = 0; i <= 30; ++i)  
{double x = i/30.0 + 1.0;  
cout  
<< x << ' '  
<< sin(x) << ' '  
<< cos(x) << ' '  
<< tan(x) << ' '  
<< log(x) << ' '  
<< exp(x) << ' '  
<< sqrt(x) << '\n';  
}  
}

Se l'avete eseguito avrete osservato senz'altro come la lista dei valori scritti non sia incolonnata affatto e somigli un po' a degli appunti presi a mano e frettolosamente. Questo è quanto ritenuto sgradevole nel modo di operare di *<<*; e ringraziate che almeno vi abbia messo gli spazi e le andate a capo, altrimenti se ne usciva così:  
  
10.8414710.5403021.5574102.7182811.033330.859010.5119581.677890.0327...   
  
Era solo l'inizio...be', intanto imparate che due stringhe virgolettate giustapposte una accanto all'altra equivalgono a una sola stringa virgolettata che le cumuli insieme; questo agevola la scrittura nel codice di stringhe molto lunghe...  
  
Ora eseguite questa variante:

# include <iostream>  
# include <iomanip>  
# include <cmath>  
using namespace std;  
  
int main(  )  
{  
cout  
<< "Questo programma scrive 31 valori "  
"di comuni funzioni elementari di "  
"una variabile\nequidistribuita "  
"nell'intervallo [1, 2].\n\n"  
"x"  
" "  
" "  
" "  
" "  
" "  
" "  
" "  
"sin(x)"  
" "  
" "  
" "  
" "  
" "  
"cos(x)"  
" "  
" "  
" "  
" "  
" "  
"tan(x)"  
" "  
" "  
" "  
" "  
"log(x)"  
" "  
" "  
" "  
" "  
"exp(x)"  
" "  
" "  
" "  
" "  
"x^0.5\n\n";  
for(int i = 0; i <= 30; ++i)  
{double x = i/30.0 + 1.0;  
cout << setprecision(3) << fixed  
<< x << setprecision(5)  
<< setw(10) << sin(x)  
<< setw(10) << cos(x)  
<< setw(11) << tan(x)  
<< setw(10) << log(x)  
<< setw(10) << exp(x)  
<< setw(10) << sqrt(x) << '\n';  
}  
}

Vista la differenza? Qui la proprietà della giustapposizione delle stringhe è stata sfruttata oltre il lecito per evitare che il *browser* faccia collassare in uno solo gli spazi consecutivi che vi sono inseriti: nella quotidianità della vita sarebbe stato sufficiente che, ad esempio, tra la parola *sin(x)* e la parola *cos(x)*, inserite nella STESSA stringa, ci fossero stati cinque spazi adiacenti per ottenere l'effetto di incolonnamento anche per i titoli della tabella.  
I veri protagonisti, tuttavia, sono i cosiddetti *manipolatori* di *output stream* ossia quelle funzioni, dichiarate nel *namespace std* e fruibili grazie all'inclusione del documento *iomanip*, tra le quali sono state scelte quelle inserite, e quindi invocate, nel presente codice, ossia *setprecision*, *setw* e *fixed*: per quest'ultima, essendo invocata senza argomenti, deve essere eccezionalmente **omessa** anche la coppia di parentesi vuote, quando si trova nel contesto dell'inserimento, ossia a destra di *<<*. Ciò è spiegabile in base alla definizione dell'*overload* della funzione associata a *<<* quando il suo operando sinistro è uno *stream* e il destro un puntatore a funzione.  
  
L'azione dei tre manipolatori introdotti nel codice è semplice: il manipolatore *setw* richiede che l'oggetto che sarà inserito nello *stream* subito dopo il suo intervento, ed ecco perché è ripetuto prima di ogni inserimento, dal secondo in poi, **occupi comunque un numero di *bytes* almeno pari al parametro trasmessogli**, anche nel caso ne potesse occupare di meno; all'interno di quello spazio il valore effettivamente scritto apparirà allineato a destra, fermo restando che, qualora dovesse occupare più spazio di quello indicato, verrà comunque scritto interamente e non di certo amputato per farcelo rientrare, a costo, però, di perdere l'effetto desiderato; occorre quindi trasmettergli valori sufficienti....  
Questa caratteristica è stata sfruttata appunto per produrre l'incolonnamento, anche giocando sul valore maggiore trasmesso a *setw* in occasione della scrittura del valore della tangente, che, per natura, raggiunge ordini di grandezza superiori a 1 nell'intervallo cui è stata fatta appartenere la variabile indipendente.  
  
L'azione di *setw* non è però sufficiente, da sola, a produrre l'effetto; occorre anche la cooperazione di *fixed*, che richiede, fino a nuovo ordine, di mantenere la scrittura degli zeri finali della parte decimale di un numero, vale a dire, ad esempio, 1.000 invece di semplicemente 1: provate a toglierlo e vedrete che cosa succede.  
  
Infine *setprecision* utilizza il parametro trasmessogli, intero, ovviamente, così come quello di *setw*, per stabilire, globalmente e fino a nuovo ordine, quante cifre saranno da scrivere **DOPO IL PUNTO** decimale, con arrotondamento automatico dell'ultima cifra scritta. Non bisogna esagerare, perché il numero di cifre affidabili è comunque finito: provate a mettere *setprecision(30)* al posto di *setprecision(3)*; il compilatore lo accetta, ma VEDRETE che quello che sarà scritto dopo la quattordicesima/quindicesima cifra è PURA FANTASIA, anche quelle poche volte che sembrerà esatto....  
  
Esistono numerosi altri manipolatori, che sono descritti in una pagina dedicata, e ne esistono anche per gli *input stream*: tra questi ultimi i soli che rivestono qualche utilità immediata, a meno di operazioni sofisticate e poco scientifiche, sono quelli che consentono di far leggere numeri interi che siano scritti in OTTALE o in ESADECIMALE, ovvero di scriverli in tali basi numeriche: ad esempio l'espressione

std::cout << 80 << ' ' << std::oct << 80 << ' ' << std::hex << 80 << '\n';

fa comparire sul terminale, a meno di reindirizzamento, il numero 80 (OTTO DECINE) prima in base 10, poi in base 8, poi in base 16; e se al posto di *cout* ci fosse *cin*, al posto di *<<* ci fosse *>>*, al posto della costante 80 tre variabili dichiarate *int* e venissero eliminate **TUTTE LE COSTANTI CARATTERE**, ossia se si scrivesse

int i, j, k;  
std::cin >> i >> std::oct >> j >> std::hex >> k;

il *Pongo pygmaeus* (orango del Borneo) alla tastiera dovrebbe ricordarsi di digitare il primo numero in decimale, il secondo in ottale e il terzo in esadecimale, altrimenti O il programma capisce fischi per fiaschi O addirittura verrà acceso *failbit*; provate per credere, senza offesa per gli oranghi.  
  
Il linguaggio, che è un GRAN linguaggio, offre infine allo *scriptor callidus* la possibilità di crearsi lui/lei stesso/a dei PROPRI manipolatori per le più varie evenienze: tali, che so, da mandare sullo *standard output stream*, e SOLO su quello, i dati a colori, o sottolineati, o a caratteri lampeggianti e magari in rosso se sono inviati da *cerr* (e SOLO da lui) e altre piacevolezze e/o bagatelle del genere, limitate solo dal livello di fantasia che si è attinto.  
  
**Nota a pie' di pagina**  
  
In un'espressione/dichiarazione del tipo  
  
*const char \* p = x . str(  ) . data(  );*   
  
ove *x* è un oggetto della classe *ostringstream* la durata della vita del puntatore a *char* restituito dal metodo *data(  )* è limitata all'espressione in cui si trova e quindi si estingue al punto e virgola finale, rendendo di fatto il puntatore *p* NON INIZIALIZZATO CORRETTAMENTE. La causa di ciò risiede nel fatto che l'oggetto che esegue il metodo *data(  )*, ossia quello che sta a sinistra dell'operatore punto, vale a dire *x . str(  )*, È PRIVO DI NOME e quindi di INDIRIZZO PERMANENTE IN MEMORIA. È LUI, in sostanza, a morire al punto e virgola, e assieme a lui muoiono, coerentemente, TUTTI I SUOI METODI e TUTTI GLI INDIRIZZI DA ESSI RESTITUITI.

**Canto ventiduesimo: il sistema ios per l'INPUT/OUTPUT**

Questo è un canto breve, scritto per presentare quegli *stream* che cumulano in sé entrambe le funzionalità dell'INPUT e dell'OUTPUT e nei cui aggregati sta quindi TUTTO quello che è stato scritto nei due canti precedenti. Il canto riprenderà anche il sospeso riguardante il metodo *putback*, che pertiene agli *stream* di INPUT, ma trova la sua migliore utilizzazione negli *stream* della presente categoria.   
  
Si tratta del genere di *stream* che usa, ad esempio, il vostro *editor di testo*, il quale ha sia bisogno di leggere il contenuto dei vostri documenti, ma anche di scriverci dentro per apportare le correzioni e/o le aggiunte e/o le rimozioni che desiderate. È pur vero che tutto potrebbe essere gestito in memoria, ma per documenti molto grandi questa potrebbe essere una controindicazione rispetto alla snellezza di uno *stream* bidirezionale (in realtà l'approccio usato di fatto è un'ibridazione dei due, sempre alla ricerca del miglior compromesso).  
  
Va da sé, si suppone, che, trattandosi in sostanza di unione di cose già dette, questi *stream* si distingueranno da quelli già presentati SOLO al momento della dichiarazione degli oggetti gestori E dell'apertura dello *stream*. In effetti i nomi dei tipi da usare sono:

**fstream** (per gli *stream* su disco)  
**stringstream** (per gli *stream* in memoria)

vale a dire gli stessi nomi visti finora, privati delle loro iniziali, e dichiarati negli stessi documenti da includere all'inizio del testo del programma.  
  
Trattandosi di *stream* tuttofare appare ovvio che non si possa più fare assegnamento su sottintendimenti di alcuna sorta, il che vuol significare che, al momento dell'inizializzazione dell'oggetto, che continua a doversi compiere nei modi ormai consueti, (quasi) nulla più potrà essere sottaciuto.  
  
Ecco dunque di seguito alcune inizializzazioni, tra le più usuali, per un oggetto *fstream*, debitamente commentate; effettuarle non contestualmente con la dichiarazione non è vincolante.

1. *fstream f;   
   f . open("pappo", ios :: in);*  
     
   questo equivale *in toto* a *ifstream f("pappo");*, ossia all'apertura di un documento su disco (di nome *pappo*) per la SOLA LETTURA [ecco perché esiste *ifstream*].
2. *fstream f;   
   f . open("pappo", ios :: out);*  
     
   questo equivale *in toto* a *ofstream f("pappo");*, ossia all'apertura di un documento su disco (di nome *pappo*) per la SOLA SCRITTURA [ecco perché esiste *ofstream*].
3. *fstream f;   
   f . open("pappo", ios::in | ios :: out);*  
     
   ecco finalmente quello per cui si scrive il presente canto, ossia l'apertura di un documento su disco (di nome *pappo*) sia per la LETTURA sia per la SCRITTURA; in questa modalità il documento di nome *pappo* DEVE TUTTAVIA PREESISTERE all'esecuzione del programma, pena accensione di *failbit*. Lo *stream* appena aperto si trova posizionato all'inizio, tanto per la lettura quanto per la scrittura (pensate agli *editor di testo*). Quello che di fatto accadrà dipende dalle due volontà del programmatore e dell'esecutore.
4. *fstream f;   
   f . open("pappo", ios::in | ios :: out | ios :: trunc);*  
     
   come sopra, ma stavolta il contenuto precedente (eventuale) di *pappo* viene spazzato via volontariamente, di modo che questa modalità di apertura dello *stream* funziona anche se *pappo* non esiste. Naturalmente la prima operazione che potrà essere portata a buon fine dovrà PER FORZA essere un'operazione di SCRITTURA.
5. *fstream f;   
   f . open("pappo", ios::in | ios :: out | ios :: app);*  
     
   come nel caso 3., ma stavolta lo *stream*, appena aperto, è posizionato all'INIZIO, per la prima operazione di LETTURA, e alla FINE (di volta in volta aggiornata), per TUTTE le operazioni di SCRITTURA. Quello che accadrà per le (eventuali) operazioni di lettura successive alla prima è nelle facoltà del programmatore.
6. *fstream f;   
   f . open("pappo", ios::in | ios :: out | ios :: ate);*  
     
   come nel caso 5., ma stavolta lo *stream*, appena aperto, è posizionato alla FINE solo per la PRIMA operazione di scrittura. Quello che accadrà per le (eventuali) operazioni, sia di lettura sia di scrittura, successive alle prime, è nelle facoltà del programmatore.

Ed ecco un analogo elenco di possibili inizializzazioni, PUNTO su PUNTO, di un oggetto *stringstream*, in cui occorre notare che stavolta è necessario effettuarle contestualmente con la dichiarazione, poiché il metodo *str* non prevedrebbe la ricezione di un secondo argomento con funzioni analoghe al secondo argomento di *open*:

1. *stringstream s("abcdefg", ios :: in);*// equivalente a *istringstream s("abcdefg");*// o anche a  
   // *istringstream s;*// *s . str("abcdefg");*
2. *stringstream s("", ios :: out);*// equivalente a *ostringstream s;*
3. *stringstream s("abcde", ios :: in | ios :: out);*// equivalente al punto 3. di *fstream*  
   // e anche, semplicemente, a *stringstream s("abcde");*
4. come il punto precedente (*ios :: trunc* è irrilevante per gli oggetti *stringstream*).
5. come il punto successivo (*ios :: app* EQUIVALE a *ios :: ate* per gli oggetti *stringstream*)
6. *stringstream s("abcde", ios :: in | ios :: out | ios :: ate);*// equivalente al punto 6. di *fstream*

Gli elenchi proposti esauriscono le combinazioni delle costanti da *ios::in* a *ios::ate* che rivestano qualche utilità e/o interesse; va tenuto conto che il compilatore accetterebbe anche una qualsiasi altra combinazione tra le 32 totali possibili, semplicemente ignorando TUTTO quel che sarebbe privo di senso e/o superfluo.  
Se però si usa una combinazione astrusa, o assurda, o comunque contraddittoria rispetto alle proprie inconsapevoli intenzioni, POI non ci si lagni se il programma non funziona come si vorrebbe.  
  
Per la verità esiste una sesta costante, che porterebbe a 64 il totale delle possibili combinazioni, denominata *ios::binary*: non vi servirà **MAI** a meno che non dobbiate lavorare in C++ su una ciofeca di piattaforma che preferisco non nominare.  
  
Qualcosa in più occorre invece dire su alcuni dettagli che riguardano l'uso di *stream* bidirezionali, e specialmente su alcune differenze sostanziali che intercorrono fra gli oggetti *fstream* e quelli *stringstream*. Gli uni e gli altri dispongono sia dei metodi *tellg*, *seekg* sia di *tellp*, *seekp*, tutti con le dichiarazioni e definizioni già note; tutti dispongono altresì del metodo *putback*, che riceve un SOLO argomento di tipo *char* e la cui discussione era stata, a suo tempo, rinviata: è giunto il momento di darla.  
  
Come detto, *putback* è pertinente a quattro tipi di oggetti (per quanto ne sapete finora):

* ***ifstream***
* ***istringstream***
* ***fstream***
* ***stringstream***

L'interpretazione corrente dello standard del linguaggio recita (traduzione la più fedele possibile dal testo originale inglese) che *putback* **"reimmette il carattere *c*** (quello che riceve come argomento, *n.d.t.*) **nell'*input stream*, di modo che il prossimo carattere a essere estratto sia *c*"**.  
  
Questa frase, secondo il modesto, ma giusto, parere dell'autore di queste note, è fuorviante (del resto che cosa ci si può aspettare dalla lingua inglese?), perché dà l'impressione che ciò accada, e BASTA, senza ulteriori discussioni, tenendo quindi nascoste le notevoli differenze di comportamento dei diversi oggetti gestori, dovute sia alla diversa natura dei supporti sia alla diversa direzionalità del flusso dei dati (ecco quello che può la lingua nostra [endecasillabo adattativo del celeberrimo verso del Divino Poeta, fatto pronunciare a Sordello mantovano quando incontra il conterraneo Virgilio]).  
  
Venendo al sodo, se *putback* viene eseguito da un oggetto gestore di *stream* con supporto su disco (leggi: *ifstream* o *fstream*), l'operazione di reimmissione nello *stream* va a buon fine SOLO SE CIÒ NON COMPORTA UNA MODIFICA DEL CONTENUTO DEL SUPPORTO, e questo ANCHE se l'oggetto è *fstream* ed è stato aperto con la clausola *ios :: in | ios :: out*. In questo senso l'azione di *putback* diventa IDENTICA a quella di *unget* (tornate a vederla), il valore del parametro trasmesso serve solo ad assicurarsi della liceità dell'azione, e la frase tradotta dall'inglese ha un significato chiaro ed esatto.  
  
Se, al contrario, l'azione di reimmissione risultasse in una modifica del contenuto del supporto, tale azione NON VA AFFATTO A BUON FINE e il *"prossimo carattere a essere estratto"* NON SARÀ AFFATTO quello trasmesso a *putback*, ad onta della recitazione di cui sopra, ma potrà MOLTO VEROSIMILMENTE tradursi nell'accensione di qualche segnalatore di errore.  
  
Purtroppo, allo stato attuale del compilatore GNU, e fors'anche dello stesso standard del linguaggio, NON C'È un modo DIRETTO per verificare se *putback* sia riuscito o no a fare quel che gli era stato richiesto, se non quello di controllare l'eventuale fallimento della successiva operazione di lettura; l'unico *escamotage* ideato dall'autore di queste note (che ha anche inoltrato [novembre 2013] una formale richiesta di chiarimento/ausilio agli sviluppatori del compilatore) consiste nel controllare quanto restituito da *tellg* SUBITO DOPO l'esecuzione di *putback* verificando che NON SIA maggiore di quanto restituiva SUBITO PRIMA (il che però implica dover sempre chiamare DUE VOLTE *tellg* per OGNI esecuzione di *putback*, oppure, **MEGLIO**, NON USARE MAI *putback* con questo genere di oggetti).  
  
Per persuadersi della veridicità delle affermazioni fatte nell'ultimo capoverso basta eseguire il seguente codicillo, una prima volta così com'è, e una seconda volta avendolo ricompilato dopo aver obbedito al commento posto subito dopo l'invocazione della funzione *system*, la quale non fa altro che far eseguire al calcolatore il comando fra virgolette come se venisse digitato sulla *shell* (GRANDI SISTEMA OPERATIVO E LINGUAGGIO). Per inciso, se aveste nella vostra cartella un documento di nome *file\_del\_piffero*, è giusto che lo PERDIATE, come di fatto avverrà qualora lo aveste.

# include <iostream>  
# include <fstream>  
# include <cstdlib>  
using namespace std;  
  
int main(  )  
{  
system("echo abcde > file\_del\_piffero");  
// sostituire abcde con abade  
fstream is("file\_del\_piffero", ios :: in | ios :: out);  
char c, d = 0;  
streampos s;  
is . get(c);  
clog  
<< "dopo get siamo al byte "  
<< is . tellg(  )  
<< " con c = " << c  
<< '\n';  
// ora si avanza di due byte lungo lo stream  
is.seekg(2, ios::cur);  
clog  
<< "dopo seekg siamo al byte "  
<< (s = is . tellg(  )) << '\n';  
is . putback(c);  
if(is && is . good(  ))  
clog << "a quanto risulta... parrebbe tutto bene ... ";  
try  
{// si controlla se l'indicatore di lettura  
// è effettivamente retrocesso di un byte  
if(is . tellg(  ) >= s) throw 0;  
}  
catch(int i)  
{cerr << "ahi! ahi! ahi!\n";  
return 222;}  
cout  
<< "\ndopo putback siamo al byte "  
<< is . tellg(  )  
<< " e il prossimo carattere estratto sarà ["  
<< (char)(d = is.get(  )) << "]\n";  
}

Quando è un oggetto *istringstream* a eseguire *putback* le cose vanno allo stesso modo per quanto riguarda la riuscita dell'operazione; in questo caso, però, l'oggetto SEGNALA la non riuscita, sicché risulta possibile effettuare un controllo DIRETTO in tal senso, come nella seguente variante del codice precedente (eseguite anche questo, nelle due diverse situazioni di contenuto iniziale dello *stream*):

# include <iostream>  
# include <sstream>  
using namespace std;  
  
int main(  )  
{  
istringstream is("abcde");  
// sostituire abcde con abade  
char c, d = 0;  
is . get(c);  
clog  
<< "dopo get siamo al byte "  
<< is . tellg(  )  
<< " con c = " << c  
<< '\n';  
// ora si avanza di due byte lungo lo stream  
is.seekg(2, ios::cur);  
clog  
<< "dopo seekg siamo al byte "  
<< is . tellg(  ) << '\n';  
is . putback(c);  
if(is && is . good(  ))  
clog << "a quanto risulta... parrebbe tutto bene ... ";  
try  
{// si controlla se l'indicatore di lettura  
// è effettivamente retrocesso di un byte  
if(is . bad(  )) throw 0;  
}  
catch(int i)  
{cerr << "ahi! ahi! ahi!\n";  
return 222;}  
cout  
<< "\ndopo putback siamo al byte "  
<< is . tellg(  )  
<< " e il prossimo carattere estratto sarà ["  
<< (char)(d = is.get(  )) << "]\n";  
}

Infine, quando si utilizza un oggetto *stringstream* aperto con la clausola *ios::in|ios::out*, l'esecuzione di *putback* va SEMPRE a buon fine e pertanto si conclude che il metodo *putback* andrebbe usato preferibilmente SOLO con questo tipo di oggetti.  
  
Si pone termine al canto sottolineando che la disponibilità simultanea dei metodi *tell* e *seek* sia con la *g* sia con la *p* finali ha le seguenti implicazioni:

* se lo *stream* ha supporto su disco (leggi: è del tipo *fstream*), anche se esplicitamente inizializzato sia per la lettura sia per la scrittura (leggi: è stato dichiarato e/o aperto specificando *ios::in|ios::out*) i metodi che terminano con *g* o con *p* sono, tra loro, COMPLETAMENTE INTERCAMBIABILI, il che significa che viene mantenuto un UNICO indicatore, valido sia per la lettura sia per la scrittura e comune a entrambe le operazioni. Per questa ragione **è ALTISSIMAMENTE AUSPICABILE**, anche se il compilatore non lo esige, che **OGNI VOLTA CHE SI ATTUA UN CAMBIAMENTO DI VERSO NEL FLUSSO DEI DATI** (vale a dire: se si vuole leggere DOPO che si è scritto, o VICEVERSA) la nuova operazione sia fatta precedere dalla richiesta di esecuzione di uno (qualsiasi) dei due metodi *seek*;
* al contrario, se lo *stream* ha supporto nella memoria del programma (leggi: è del tipo *stringstream*), **VENGONO MANTENUTI DISTINTI E SEPARATI DUE** indicatori, UNO per la lettura (gestito dai metodi il cui nome termina con 'g') e UN ALTRO per la scrittura (gestito dai metodi il cui nome termina con 'p').

Il codice seguente esemplifica quanto appena detto (giovarsi dei commenti inseriti):

# include <iostream>  
# include <fstream>  
# include <sstream>  
using namespace std;  
  
int main(  )  
{  
/\*  
questo programma si prefigge lo scopo di sovrascrivere  
il PRIMO byte di un documento su disco col valore  
dell'ultimo byte che vi si trova. Per conseguirlo, e per  
fini dimostrativi, utilizza TANTO un input/output stream  
con supporto sul documento stesso QUANTO un input/output  
stream con supporto in memoria.  
\*/  
// PARTE PRIMA: dichiarazione degli stream su disco.  
// INPUT stream sul documento:  
ifstream i("file");  
// I/O stream sullo stesso documento  
fstream io("file", ios :: in | ios :: out);  
char \* dati; // buffer per i dati su disco  
// PARTE SECONDA  
// (inizializzazione del puntatore dati):  
streamsize s;  
i . seekg(0, ios :: end),  
dati = new char[s = i.tellg(  )],  
i . seekg(0),  
i . read(dati, s),  
i . close(  );  
// PARTE TERZA  
// (dichiarazione dell'oggetto stringstream):  
stringstream is(dati);  
// PARTE QUARTA: correzione diretta del documento  
/\*  
posizionamento dell'indicatore di lettura  
all'ULTIMO byte (l'indicatore di scrittura LO SEGUE):  
\*/  
io . seekg(-1, ios :: end);  
char c;  
/\*  
LETTURA dell'ULTIMO byte:  
\*/  
io . get(c);  
/\*  
A QUESTO PUNTO, entrambi gli indicatori (lettura e scrittura)  
si trovano OLTRE l'ultimo byte. Se ORA si provasse a leggere  
si accenderebbe eofbit; se si provasse a scrivere si estenderebbe  
lo stream; piuttosto si riportano ENTRAMBI gli indicatori all'inizio:  
\*/   
io . seekp(0);  
// e vi si DEPOSITA c  
io . put(c);  
io . close(  );  
// ALLA FINE DEL PROGRAMMA SI CONTROLLI  
// IL CONTENUTO DEL DOCUMENTO  
// ORA SI FA LA STESSA COSA CON L'OGGETTO is  
// si posiziona l'indicatore di LETTURA all'ultimo byte  
// (quello di scrittura NON lo segue)  
is . seekg(-1, ios :: end),  
// si LEGGE l'ultimo byte  
is . get(c);  
// e lo si SPARA direttamente nello stream  
is . put(c);   
// si sarebbe addirittura potuto CONDENSARE  
// le precedenti due linee in questa sola:  
// is . put(is.get(  ));  
/\*  
ORA SI VA AD APPENDERE ALLO STREAM IL byte NULLO  
\*/  
is . seekp(0, ios :: end), is . put(0);  
/\* e si mostra su output il contenuto dello  
stream: \*/  
cout << is . str(  ) . data(  );  
}

**Quinta pausa di riflessione**

Questa è una VERA pausa di riflessione, nel senso che non vi verrà proposto alcun esercizio ma piuttosto rivolta una domanda molto SERIA, alla quale dovreste rispondere con la pienezza dell'onestà e maturità di cui la vostra autocoscienza è capace:

quanto avete capito e trattenuto nel cervello finora, in cifra percentuale?

se la risposta che darete a voi stessi, sotto le condizioni richieste e nel segreto della vostra stanza, fosse: "meno dell'80%" **NON PROSEGUITE LA LETTURA**: scegliete senz'altro il sottostante collegamento che esorta a ricominciare daccapo, ponendovi al riparo sotto l'usbergo della prima e, soprattutto, della terza virtù del *buon programmatore*. Ovvero rivolgetevi personalmente al vostro docente manifestando apertamente e senza nascondimenti il vostro disagio.  
**Chi prosegue da questo punto lo faccia con consapevolezza e assumendosene il rischio**.

**Canto ventitreesimo: creazione di oggetti: Parte I**

*O voi che siete in piccioletta barca,  
desiderosi d'ascoltar, seguiti  
dietro al mio legno che cantando varca,  
  
tornate a riveder li vostri liti:  
non vi mettete in pelago, ché forse,  
perdendo me, rimarreste smarriti.  
  
L'acqua ch'io prendo già mai non si corse;  
Minerva spira, e conducemi Apollo,  
e nove Muse mi dimostran l'Orse.  
  
Voialtri pochi che drizzaste il collo  
per tempo al pan de li angeli, del quale  
vivesi qui ma non sen vien satollo,  
  
metter potete ben per l'alto sale  
vostro navigio, servando mio solco  
dinanzi a l'acqua che ritorna equale.  
  
Que' glorïosi che passaro al Colco  
non s'ammiraron come voi farete,  
quando Iasón vider fatto bifolco.*  
  
I Divini Versi con cui il Poeta apre il secondo canto del Paradiso sono quanto di più appropriato a descrivere quel che vi aspetta salpando le ancore da questo ormeggio, quantunque possa sembrare un po' azzardato, trattandosi della presentazione di un linguaggio di programmazione, parlare di *"acqua che mai non si corse"*...  
In ogni caso, se avete *"drizzato per tempo il collo al pan"* servitovi finora, e siete *"desiderosi d'ascoltar"*, seguite e issate la vela della vostra *"piccioletta barca"*: vedrete che si gonfierà e vi porterà dove davvero *"vi ammirerete"* almeno quanto gli Argonauti vedendo *"Iasón fatto bifolco"*.  
  
Veniamo al vento: parlare di creazione di oggetti è parlare di qualcosa di ontologicamente demiurgico. Fin qui si sono incontrati numerosi oggetti (fin dall'inizio si è denominato *cout* con tale nome), ma tutti erano stati creati da qualcun altro e ce li si ritrovava belli e pronti per l'uso; per la verità, più recentemente, si era provveduto a crearne qualcuno, ogni volta che si dichiarava una variabile (oggetto) appartenente ai tipi incontrati durante le lunghe discussioni sui problemi di INPUT e OUTPUT. Anche in questi casi, però, erano i tipi degli oggetti a essere stati creati da altri e quello che si faceva consisteva semplicemente nell'istanziazione (la realizzazione) di un oggetto il cui *progetto*, la cui *idea*, non ci apparteneva. Il *demiurgo* non è tanto il realizzatore materiale di un oggetto (per questa mansione basta un manovale abbastanza abile: siete bastati voi per realizzare un *ifstream*), quanto l'ideatore del progetto di quell'oggetto, ossia chi ha definito che cosa sia un *ifstream* e come sia fatto.   
  
È giunta l'ora di disporsi NON a creare gli oggetti, ma a PROGETTARLI, ossia a creare la loro *IDEA PLATONICA*, in modo che tanto il progettista stesso quanto CHIUNQUE ALTRO possa realizzarne quanti SINOLI ARISTOTELICI se ne vogliano (l'unica cosa che si sia fatta finora; vi ho detto o no di issare la vela?).   
  
Ogni idea che nasce in una mente razionale prende forma e si sostanzia in relazione con idee preesistenti. Non è questa la sede per disquisire su quali idee siano da considerare innate e quali siano idee derivate da quelle: ci basta accogliere come valida questa distinzione, suffragata peraltro, in ossequio al metodo scientifico galileiano, da numerose osservazioni sperimentali oggettive.   
  
Si pensi, ad esempio, all'idea di *studente\_di\_fisica*; non sarà difficile riconoscere che tale idea deriva da quella, più generale, di *studente*, che a sua volta si concepisce relativamente ad altre idee che, risalendo la corrente delle deduzioni, potrebbero giungere fino all'idea primigenia di *essere\_umano*... e potrebbe perfino non essere finita lì.   
  
Una catena di deduzioni ideali del tutto simile a quella appena prospettata prende corpo in C++ nel concetto di tipi derivati uno dall'altro, quei tipi cui, si è detto, stiamo per dare esistenza agli occhi del compilatore.  
  
Lo strumento tecnico di quest'opera demiurgica risiede nelle parole di vocabolario *enum* (già vista), *union* (la cui conoscenza può essere rinviata e demandata alla lettura della pagina ad essa dedicata) e, **SOPRATTUTTO**, *struct* e *class*, che sono totalmente interscambiabili tra loro, eccetto che per un particolare che apparirà chiaro ben presto.  
  
E, già che ci siamo, nel prossimo canto ci si introdurrà nell'universo delle idee, prendendo giustappunto come campione quella dello *studente\_di\_fisica*, nella quale non dovreste far fatica a immedesimarvi, e la considereremo nelle sue derivazioni gerarchiche da quella, immediatamente antecedente, di *studente* e da quella, più remota, di *essere\_umano*.   
  
Dovrebbe apparire subito evidente che le qualità caratterizzanti un *essere\_umano* debbano potersi ritrovare (almeno fino a prova contraria) in uno *studente* e quelle di uno *studente* in uno *studente\_di\_fisica*, mentre non è vero il viceversa, ossia, in generale, NON È DETTO che le qualità caratterizzanti uno *studente\_di\_fisica* appartengano a qualsiasi *studente* o perfino a OGNI *essere\_umano*.

**Canto ventiquattresimo: creazione di oggetti: Parte II**

Si entrerà ora nella discussione dettagliata sul modo in cui quanto detto nel precedente canto si cali concretamente nella scrittura di un programma, e si procederà *ab ovo*, come si è fatto fin dall'inizio.   
  
Per manifestare al compilatore l'intenzione di creare il tipo *studente\_di\_fisica*, dotato delle relazioni introdotte nel canto scorso, basta questo programma minimale, già compilabile ed eseguibile, pur essendo privo di qualunque utilità:

# include <iostream>  
using namespace std;  
  
class essere\_umano;  
class studente;  
class studente\_di\_fisica;  
  
class essere\_umano{ };  
class studente : essere\_umano{ };  
class studente\_di\_fisica : studente{ };  
  
int main( )  
{studente\_di\_fisica amilcare, teodolinda;  
clog   
<< "sono stati realizzati due *\"studente\_di\_fisica\"*\n";}

Le prime tre righe diverse dal solito sono dichiarazioni anticipate di tipi, le successive tre sono le loro definizioni complete. Nel programma proposto le dichiarazioni anticipate sono superflue; eliminatele e vedrete che il programma si compilerà ugualmente. È esattamente come accade per la superflua dichiarazione, senza definizione, di una funzione quando immediatamente dopo comparisse la definizione della stessa funzione. Tuttavia, diversamente da quanto avviene per le funzioni, le dichiarazioni anticipate di tipo possono divenire assolutamente indispensabili in determinate situazioni che si vedranno; per giunta la loro presenza nel programma proposto gli conferisce il pregio di evidenziare che le eventuali relazioni di dipendenza di un tipo rispetto a un altro vanno espresse solo all'atto della definizione e MAI in una dichiarazione anticipata; il codice ha, inoltre, anche la virtù di mostrare COME simili relazioni di dipendenza vadano espresse.  
  
Già solo per il quasi niente che è stato scritto, *main* può affermare quanto inserisce nello *standard error stream*, senza tema di essere accusata di mentire: *amilcare* e *teodolinda* sono, a pieno titolo, due oggetti di tipo *studente\_di\_fisica* e come tali sono stati dichiarati nell'ambito di *main*, la quale vede la definizione completa di tale tipo dato che è avvenuta nell'ambito globale. Se questo non vi fosse ancora palese, tornate a leggere dall'inizio tutto il percorso.  
  
L'inutilità di un tipo così frugale da essere costituito da un aggregato vuoto è perfettamente paragonabile con quella del primo *main* che sia stato scritto in queste note: voi dovreste essere ormai abbastanza scafati, anche perché avete già usato le classi per la gestione degli *stream*... (per inciso, **DI QUESTO** si trattava, dovreste averlo intuito...) per non capire che qualcosa dovrà essere scritto tra le graffe che definiscono la classe, e anche per intuire che, non appena questo si farà, le cose si complicheranno.  
  
Tanto per procedere piano, e con gradualità, si cominci col supporre che *amilcare* e *teodolinda* dovranno pur avere un nome proprio, e che questa qualità è ben rappresentabile con una stringa di caratteri; questo significa che l'aggregato *studente\_di\_fisica*, che d'ora in avanti si chiamerà col suo VERO nome: *ambito della classe*, **dovrà contemplare una variabile che sia capace di contenere una stringa di caratteri** e che **ogni oggetto potrà attribuirle, come valore, il proprio nome**.  
  
**FICCATEVI NELLA TESTA LE FRASI IN GRASSETTO, PERCHÉ *QUESTO* È IL MECCANISMO CHE VALE PER OGNI ALTRA VARIABILE, DI OGNI TIPO, CHE SI INSERIRÀ NELL'AMBITO DI UNA CLASSE**.  
  
D'altra parte il fatto di avere un nome proprio NON È, con tutta evidenza, un'esclusiva di uno *studente\_di\_fisica*, che ne beneficia piuttosto in quanto *studente*, perché anche gli studenti di lettere hanno un nome, ma NON in quanto *essere\_umano* perché l'idea di essere umano precede la civilizzazione e la socializzazione, che attribuiscono nomi agli individui.  
  
Queste considerazioni suggeriscono che la variabile capace di detenere il nome vada inserita nell'ambito di *studente* e che *studente\_di\_fisica* possa riceverla in eredità, essendo definita come dipendente da tale classe.  
  
Infine sarebbe opportuno che, come avveniva ad esempio per la classe *ifstream* (ricordate?), all'atto della dichiarazione di un oggetto se ne possa, contestualmente, inizializzare il valore del nome, scrivendo la dichiarazione, sull'esempio di *ifstream*, con qualcosa di simile a *studente\_di\_fisica teodolinda("Teodolinda de Bavaris");*.  
  
Supponendo che gli ultimi tre capoversi vi siano CHIARI sul piano della LOGICA, altrimenti domandate a voce, perché non saprei in quale altro modo scriverli, ecco la realizzazione tecnica, ossia sintattica, dei loro asserti, avendo anche abolito, per ora, le dichiarazioni anticipate, per minimizzare il superfluo:

# include <iostream>  
# include <string>  
using namespace std;  
  
class essere\_umano{ };  
class studente : essere\_umano  
{protected:  
string nome;  
public:  
studente(const char \* s) : nome(s) { }};  
  
class studente\_di\_fisica : studente  
{public:  
studente\_di\_fisica(const char \* s) : studente(s) { }  
};  
  
int main( )  
{studente\_di\_fisica  
amilcare("Amilcar de Chartaginensibus"),  
teodolinda("Teodolinda de Bavaris");  
clog   
<< "sono stati realizzati due *\"studente\_di\_fisica\"*\n";}

**AVETE COMPILATO IL PROGRAMMA?** Eccovi la spiegazione delle complicazioni che sono state necessarie:

1. nell'ambito di una classe vanno inserite dichiarazioni di variabili/oggetti e di funzioni, queste ultime non obbligatoriamente con definizione contestuale. Oggetti e funzioni, chiamate nel presente contesto metodi, sono detti collettivamente membri della classe;
2. una classe è considerata un tipo completo, ossia tale da consentire la creazione di un oggetto, quando NON HA NULLA DI NON DEFINITO, non quando NON HA NULLA e basta, perché si è già visto che, in tal caso è considerata completa: completa di nulla, ma completa. La frase sottolineata significa che devono essere state definite tutte le funzioni eventualmente solo dichiarate entro l'ambito, e devono essere state completate ANCHE tutte le classi da cui eventualmente erediti qualcosa o cui eventualmente appartenga qualche variabile dichiarata nel suo ambito;
3. l'annunciata variabile detentrice del nome proprio è stata opportunamente chiamata *nome* ed è stata dichiarata, nell'ambito di *studente*, come appartenente al tipo *string*, per il cui uso è stato necessario includere all'inizio l'omonimo documento: *string* è, ANCHE LEI, una classe, ivi dichiarata e COMPLETATA, il che assicura la completezza ANCHE di *studente*, per quanto si è scritto sopra. Tale classe consente la gestione delle stringhe di caratteri in modo molto più efficiente di un semplice contenitore di *char*, *array* o puntatore che sia; tanto per dirne solo una non vi è ALCUN BISOGNO di preoccuparsi della lunghezza della stringa... hai detto poco...
4. sia nella classe *studente\_di\_fisica* sia nella classe *studente* sono state inserite delle etichette, che sono parole di vocabolario: *protected* e *public* in *studente*; solo *public* in *studente\_di\_fisica*. Ne esiste una terza, chiamata *private*, che è **sottintesa** in una *class*, mentre in una *struct* è *public* a essere sottintesa: **È QUESTA LA SOLA DIFFERENZA FRA *struct* E *class* CHE ERA STATA ANNUNCIATA** (ricordate?). Detto che il segno di due punti è OBBLIGATORIO, anche separato con spazi, per terminare l'etichetta nel contesto attuale, il che SIGNIFICA CHE NE ESISTERÀ ALMENO UN ALTRO: non dimenticate mai nel cassetto la vostra LOGICA, resta da dire a che cosa servono.
5. Dato che in una *class* l'etichetta *private* è sottintesa, ciò comporta che OGNI DICHIARAZIONE AVVIENE SOTTO LA SUA INFLUENZA fino a quando non si inserisce una delle altre due etichette, che prende a sua volta a influire sulle dichiarazioni successive fino ad avviso diverso; pertanto in *studente* la variabile *nome* è *protected* e tutto il resto dell'ambito è *public*, come l'intero ambito di *studente\_di\_fisica*. In sostanza, nel nostro programma, solo l'ambito di *essere\_umano*, peraltro tuttora vuoto, e quindi COMPLETO, è rimasto sotto l'influenza di *private*.
6. Come suggerisce il nome stesso delle tre etichette, i membri di una classe, e spero che si sia intuito che quando scrivo classe con questo corpo di carattere intendo l'unione di *class* e *struct*..., dichiarati sotto l'influenza di *public* SONO ACCESSIBILI A CHIUNQUE in ogni parte del programma; quelli dichiarati sotto *private* sono accessibili **SOLAMENTE** agli altri membri della stessa classe, mentre quelli dichiarati sotto *protected* sono accessibili ANCHE, ma SOLO ANCHE, ai membri delle eventuali classi eredi.
7. Per quanto appena detto, la variabile *nome* è accessibile SOLTANTO ai membri di *studente* E a quelli di *studente\_di\_fisica*; a NESSUN ALTRO.
8. Sia in *studente* sia in *studente\_di\_fisica* sono state inserite anche due funzioni molto particolari il cui nome ***COINCIDE*** con quello della classe cui appartengono e, per ciò stesso, diversamente da ogni altra funzione della galassia e dell'intero gruppo locale, **NON HANNO BISOGNO DI ESSERE DICHIARATE COME RESTITUENTI UN CERTO TIPO** perché il tipo che restituiscono si chiama esattamente come il loro stesso nome: È LA LORO CLASSE. Le funzioni così individuate sono denominate i **costruttori della classe** e ogni classe ne può possedere **un numero arbitrario** grazie al meccanismo dell'*overload*, proprio di qualsiasi funzione (GRAN LINGUAGGIO). Si osservi che entrambi i costruttori sono dichiarati e definiti nelle zone *public* delle rispettive classi; nulla vieta di definire dei costruttori anche nella zona *private*...
9. La sintassi della definizione di un costruttore non è particolare solo per il fatto di avere nome e tipo restituito collassati in un'unica parola, ma anche per la presenza **OPZIONALE** della cosiddetta lista degli inizializzatori, posta tra il segno di due punti, OBBLIGATORIO, se si mette una lista, e la graffa che apre l'ambito del costruttore. In detta lista possono trovare posto, separate con la virgola, inizializzazioni di proprie variabili, come fa il costruttore di *studente* rispetto alla propria variabile *nome*, di fatto trasmettendo l'argomento *s* ricevuto A UNO DEI COSTRUTTORI DI *string*, o invocazioni esplicite di un costruttore della classe antenata, come fa il costruttore di *studente\_di\_fisica*, invocando quello di *studente* trasmettendogli l'argomento ricevuto, o sia l'uno sia l'altro.
10. Uno dei costruttori di una classe, quello che ha la lista di argomenti adeguata, è eseguito tacitamente e AUTOMATICAMENTE all'atto stesso della dichiarazione di un oggetto come appartenente a tale classe.

Adesso respirate liberamente per due minuti e poi tornate a leggere daccapo TUTTO l'elenco puntato; dopo di che uscite, fate due passi o prendete un tè. NON FUMATE! Vi danneggia irreparabilmente il cervello! Tornate a leggerlo e così via FINO A QUANDO NON SARÀ COMPLETAMENTE ENTRATO NELLA PARTE DELLA VOSTRA ZUCCA PREPOSTA AL RAGIONAMENTO LOGICO. Nel frattempo anch'io mi riposerò un po'.  
  
Ben ritrovati, come vi sentite? Se aveste definitivamente abbandonato il fondale ove si consuma solo una vita da *Veneridae*, e foste incamminati con lena per l'alto sale, molte domande dovrebbero frullarvi in mente; tante che non potrei nemmeno contarle. Cooome? Non ne avete? Allora tornate al canto zero! E senza passare dal "Via"!  
  
A quelli che sono ancora qui proverò a rispondere; gli altri tornino quando avranno imparato a nuotare.  
  
Domanda numero 1:  
se, ex punto 1 dell'elenco, la definizione di un metodo non è obbligatoria nell'ambito di una classe, visto poi che, ex punto 2, quella stessa definizione è detta essenziale per poter affermare la completezza della classe, e quindi la sua utilità stessa, QUANDO E COME SI DEFINISCE UN METODO SOLO DICHIARATO? Possibile che non aveste QUESTA domanda?  
  
La risposta è in questa variante del programma, in cui si usa come metodo campione il costruttore stesso di *studente\_di\_fisica*:

# include <iostream>  
# include <string>  
using namespace std;  
  
class essere\_umano{ };  
class studente : essere\_umano  
{protected:  
string nome;  
public:  
studente(const char \* s) : nome(s) { }};  
  
class studente\_di\_fisica : studente  
{public:  
// costruttore SOLO dichiarato:  
studente\_di\_fisica(const char \*);  
};  
// definizione del costruttore SOLO dichiarato  
// (si usa il risolutore di ambito: la lista degli  
// inizializzatori compare SOLO   
// all'atto della definizione):  
studente\_di\_fisica :: studente\_di\_fisica(const char \* s) : studente(s) { }  
  
int main( )  
{studente\_di\_fisica  
amilcare("Amilcar de Chartaginensibus"),  
teodolinda("Teodolinda de Bavaris");  
clog   
<< "sono stati realizzati due *\"studente\_di\_fisica\"*\n";}

Domanda numero 2:  
se *string* è un tipo così efficiente per la gestione delle stringhe di caratteri, perché non è stato mai usato al posto degli *array* e/o dei puntatori a *char*?  
Risposta:  
perché non sapevate ancora che cosa fosse una classe, e anche adesso non illudetevi di saperne già abbastanza.  
  
Domanda numero 3:  
se in una *struct* è sottintesa l'etichetta *public*, perché non si usa una *struct* al posto di una *class*, così che problemi di accessibilità non ci saranno mai?  
Risposta:  
perché è sempre meglio imparare nelle condizioni più restrittive, e poi l'idea di lasciare accessibile TUTTO a cani e porci ha pure un bel numero di controindicazioni.  
  
Domanda numero 4:  
quale sarebbe quest'*almeno un altro contesto* in cui potrebbero comparire i nomi delle etichette?  
Risposta:  
nella clausola che indica la dipendenza di una classe da una o più classi antecedenti. Nel programma proposto, in mancanza di indicazioni esplicite, si sottintende ancora una volta *private*, trattandosi di *class*; vale a dire che è come se si fosse scritto:  
  
*class studente\_di\_fisica : private studente{/\* omissis \*/};*   
  
per cui la variabile *nome*, che era *protected* in *studente*, diventa *private* in *studente\_di\_fisica*; se si fosse scritto in QUELLA posizione *protected* o *public*, la variabile *nome* sarebbe rimasta *protected* anche nella classe erede. Ma su questo si tornerà.  
  
Domanda numero 5:  
ma se abbiamo scelto di usare *class*, e risulta che per tale scelta la variabile *nome* è accessibile solo ai membri delle classi *studente* e *studente\_di\_fisica*, ex punto 7 dell'elenco, visto che *main* non è membro di tali classi (o no?), come si farebbe se si volesse, ad esempio, fare scrivere su *standard output stream* il nome di uno studente?  
Risposta:  
questa è una delle domande migliori che abbiate potuto porvi; in effetti *main* non potrà MAI essere membro di veruna classe, e quindi, nel caso presente, NON HA ACCESSO a *nome*. Pertanto, allo stato attuale del programma, *main* non può scrivere il nome di chicchessia: se si volesse raggiungere un tale obiettivo occorrerebbe arricchire la *class studente\_di\_fisica* con un metodo ad accesso *public* che, in quanto membro, HA ACCESSO a *nome* e ne restituisca il valore a chi, come *main*, nella fattispecie, abbia diritto di eseguirlo: ossia CHIUNQUE, essendo, LUI, *public*. In altre parole, per accedere a membri NON pubblici, occorre sempre fornire un intermediario pubblico che faccia parte della classe, il che significa, per inciso, poter mantenere SEMPRE l'accesso ai membri della classe SOTTO IL CONTROLLO DIRETTO dell'**AUTORE** della classe stessa.   
  
Domanda numero 6:  
e quando, nella prima versione, quella con le classi vuote, il programma comunque veniva eseguito e si è affermato che non mentisse quando asseriva di aver realizzato DUE *studente\_di\_fisica*, **CHI DIAMINE LI AVEVA REALIZZATI** se non c'erano costruttori, NON C'ERA NULLA, e si afferma, ex punto 10, che *all'atto stesso della dichiarazione di un oggetto* viene eseguito un costruttore della classe? e addirittura quello che ha la lista di argomenti adeguata? Ci stai prendendo in giro?  
[ma davvero QUESTA domanda non l'avevate? Quanta sabbia avete addosso?]  
Risposta:  
Un'altra domanda intelligente, anche se, rispetto a quella sopra, denota un filino di mancanza di fiducia nel vostro docente, che MAI vi prenderebbe in giro. Il fatto è che, quando i costruttori mancano del tutto, il compilatore stesso ve ne fornisce, sua sponte, BEN TRE, sui quali, però, DOVETE ISTANTANEAMENTE SMETTERE DI FAR CONTO non appena voi cominciate a scriverne esplicitamente QUALCUNO. Se guardate attentamente, nel programma che aveva le classi vuote ci si limitava a dichiarare gli oggetti senza pretendere di inizializzarli in alcun modo, anche perché NON VI ERA NIENTE DA INIZIALIZZARE. Orbene, tra i costruttori prestati gratuitamente dal compilatore c'è quello denominato, non senza motivo, *costruttore di default* che ha appunto la lista di argomenti adeguata perché HA LA LISTA VUOTA, al netto di argomenti *standard*. Anche su questo si tornerà.  
  
Domanda numero 7:  
che cosa accade, esattamente, nella memoria del programma?  
Risposta:  
Domanda formidabile, cui risponderò dettagliatamente nel prossimo canto.   
  
Domanda numero 8:  
chi sono Amilcare e Teodolinda?  
Risposta:  
Ignoranti!  
  
Tali, più o meno, avrebbero dovuto/potuto essere, credo, le vostre domande; se ne aveste delle altre, ponetele a lezione o in laboratorio.

**Canto venticinquesimo: creazione di oggetti: Parte III**

*La sete natural che mai non sazia  
se non con l'acqua onde la femminetta  
samaritana domandò la grazia,  
  
mi travagliava, e pungeami la fretta  
per la 'mpacciata via dietro al mio duca,  
e condoleami a la giusta vendetta.*  
  
Questo, a parte il *"condolersi a la giusta vendetta"*, dovrebbe essere il vostro stato d'animo attuale. Ho promesso che avrei risposto, in questo canto, alla domanda numero 7 con cui si chiude il precedente, e lo farò, ma prima aggiungiamo ancora qualche ingrediente alla salsa del programma campione che si sta utilizzando.  
Risulta del tutto spontaneo osservare che uno *studente\_di\_fisica*, questa volta come eredità proveniente dal suo *essere\_umano*, dispone di diverse qualità caratteristiche di quella classe; tanto per citarne solo una i nostri *amilcare* e *teodolinda* sembrano essere rispettivamente un maschietto e una femminuccia, ma questo lo desume un umano che legge, non un calcolatore che capisce solo uni e zeri.  
  
D'altronde, perfino un umano, se non è turco, avrebbe difficoltà a intuire se una dichiarazione come  
  
*studente\_di\_fisica zeynep("Zeynep Yücel");*concerne un uomo o una donna: io, per esempio, l'ho imparato solo perché un mio ex-dottorando ha sposato una ragazza turca con questo nome.  
  
Ne segue che sia opportuno corredare la classe *essere\_umano* di una variabile atta a rappresentare il sesso di tale essere, e siccome il vostro docente appartiene a una scuola tradizionalista, introdurrà, alla bisogna, una variabile di tipo *bool*, la quale, per NATURA, assume solo UNO di DUE valori: poi, che il valore *true* o *false* denoti il maschio o la femmina è problema futile e del tutto irrilevante.  
  
Il problema vero, piuttosto, è un altro: se tale variabile viene inserita nell'ambito di *essere\_umano*, come appare giusto che sia, NON PUÒ PERVENIRE alla classe *studente\_di\_fisica* finché la classe intermedia, ossia *studente*, continua a essere dichiarata dipendente da *essere\_umano* nella forma usata allo stato attuale del codice (rileggetelo).  
  
Infatti si è detto (rileggete!) che l'assenza di etichette nella clausola che definisce la dipendenza di una *class* implica che sia sottintesa l'etichetta *private:*; in tal modo, però, *studente* trasforma in proprie variabili private tutte quelle che le provengono da *essere\_umano*, PERFINO se questa avesse dichiarato la variabile booleana in questione nella propria zona *public:*; e le variabili *private* non sono trasmesse ad ALCUNA CLASSE, neppure in eredità. Rileggete attentamente il canto scorso.  
  
Dato che considerare uno *studente\_di\_fisica* una sorta di entità asessuata appare del tutto inaccettabile, è evidente che occorra fare qualcosa. Non solo: come si fa a comunicare al programma che *zeynep* è una donna, già all'atto della sua dichiarazione, se la variabile a ciò deputata continuasse a essere inaccessibile a *main*?   
  
Qui è opportuno introdurre un nuovo ASSIOMA:  
 **"nella programmazione in C++, OGNI problema HA SEMPRE NUMEROSE SOLUZIONI, comunque in numero finito"**  
diversamente da quanto avviene in matematica, dove, in genere, se la soluzione di un problema non è UNICA è perché O non c'è O ce ne sono infinite.  
  
Ecco quindi di seguito UNA delle possibili soluzioni; si utilizzerà QUESTA versione del programma campione per rispondere alla domanda numero 7 del canto scorso:

# include <iostream>  
# include <iomanip>  
# include <string>  
# include <cstring>  
using namespace std;  
  
class essere\_umano  
{public:  
enum class Sesso : bool  
{maschio=true, femmina=false};  
protected:  
Sesso sesso;  
essere\_umano(Sesso b) {sesso = b;}};  
  
class studente : essere\_umano  
{protected:  
string nome;  
using essere\_umano :: sesso;  
public:  
using essere\_umano :: Sesso;  
studente(const char \* s, Sesso b) :  
nome(s), essere\_umano(b) { }};  
  
class studente\_di\_fisica : studente  
{friend ostream & operator <<  
(ostream &, studente\_di\_fisica);  
public:  
static size\_t massima\_lunghezza;  
Sesso rendiSesso() {return sesso;}  
studente\_di\_fisica  
(const char \*s, Sesso b = Sesso :: maschio) :  
studente(s, b)  
{massima\_lunghezza =   
max(massima\_lunghezza, strlen(s));}};  
  
size\_t studente\_di\_fisica :: massima\_lunghezza = 0;  
  
ostream & operator <<   
(ostream &o, essere\_umano :: Sesso s)  
{string sessi[ ]  
{"femminile", "maschile"};  
bool  
indice =  
static\_cast<bool>(s) ==   
static\_cast<bool>(essere\_umano :: Sesso :: maschio)  
&& sessi[true] == "maschile"  
||  
static\_cast<bool>(s) ==   
static\_cast<bool>(essere\_umano :: Sesso :: femmina)  
&& sessi[true] == "femminile";  
return o << sessi[indice];}  
  
ostream & operator <<  
(ostream &o, studente\_di\_fisica s)  
{o . setf(ios :: left, ios :: adjustfield);  
return  
o << setw(1+s.massima\_lunghezza)  
<< s . nome << " di sesso " << s . sesso;}  
  
int main( )  
{essere\_umano :: Sesso sesso;  
studente\_di\_fisica  
amilcare("Amilcar de Chartaginensibus"),  
teodolinda  
("Teodolinda de Bavaris",  
sesso = essere\_umano :: Sesso :: femmina),  
array\_di\_studenti[ ]  
{studente\_di\_fisica  
("Federico Barbarossa",  
sesso = essere\_umano :: Sesso :: maschio),  
studente\_di\_fisica  
("Pia de' Tolomei",  
sesso = essere\_umano :: Sesso :: femmina),  
studente\_di\_fisica  
("Gerolamo Savonarola")};  
clog   
<< "ecco gli oggetti \"studente\_di\_fisica\"\n";  
cout << amilcare << '\n' << teodolinda << '\n';  
for(studente\_di\_fisica stud : array\_di\_studenti)  
cout << stud << '\n';}

Si eseguirà ora una dettagliata analisi grammaticale e logico-sintattica di tutto il programma, non senza prima aver notato la sua assoluta neutralità rispetto ai sessi: se lo compilerete e lo eseguirete (**COMPILATELO ED ESEGUITELO!**) vedrete che, quand'anche scambiaste i valori booleani delle costanti *maschio* e *femmina*, invertiste di posizione ordinale le stesse due costanti, scambiaste di posto le due stringhe costanti *"maschile"* e *"femminile"* nell'array *sessi*...il programma risponderà IMPERTERRITO SEMPRE ALLO STESSO MODO, almeno fino a quando non gli mentirete spudoratamente trasmettendo ai costruttori informazioni FALSE, come ad esempio che *teodolinda* fosse un uomo. Una simile, pregevole, costanza di comportamento è buona testimone, pur nella semplicità di un codice come questo, di una logica ben fondata e pressoché a prova di bomba.  
  
Ci si ponga nei panni del calcolatore che si accinge a eseguire il codice binario che è stato allestito dal *linker*; come si sa l'esecuzione INIZIA dalla funzione *main*: ivi, dopo una semplice dichiarazione di una variabile *sesso* che appartiene solo all'ambito di *main* ed è del tipo *essere\_umano :: Sesso*, che vedremo al momento giusto, la **prima cosa che deve essere fatta** è la realizzazione di un oggetto chiamato *amilcare* e che è dichiarato essere di tipo *studente\_di\_fisica*.  
  
Si è già detto che la dichiarazione di un oggetto, di tipo non nativo, IMPLICA L'ESECUZIONE DI UN COSTRUTTORE della classe cui l'oggetto è fatto appartenere, e che abbia una lista di argomenti adeguata alla dichiarazione.  
  
Se si sta eseguendo il programma significa che il *linker* ha potuto trovare un costruttore idoneo alla realizzazione di *amilcare* e in effetti l'unico costruttore di *studente\_di\_fisica* riceve due argomenti, dei quali il primo è proprio del tipo presente nella dichiarazione di *amilcare* e il secondo è *standard*; quindi è adatto.  
  
Ma il compilatore, quando aveva compilato il codice, si era pur dovuto accorgere che la classe *studente\_di\_fisica* era stata fatta dipendere dalla classe *studente* e questa, a sua volta, dalla classe *essere\_umano*; pertanto comunicherà al *linker* di predisporre l'eseguibile in modo tale che, ogni volta che si dovesse istanziare un oggetto *studente\_di\_fisica*, si tenga conto delle dipendenze di questa classe. In definitiva, per costruire *amilcare*, bisogna prima aver costruito, nell'ordine, un *essere\_umano* e uno *studente*: in fondo è esattamente quello che è accaduto nella vita reale di qualsiasi *amilcare* che sia uno "studente di fisica", scritto staccato, perché in italiano.  
  
Questa, pertanto, è la sequenza cronologica delle operazioni che sono necessarie alla realizzazione di *amilcare*, e che quindi **sono realmente compiute**:

1. realizzazione di un oggetto SENZA NOME appartenente alla classe *essere\_umano*;
2. realizzazione di un oggetto SENZA NOME appartenente alla classe *studente*, che assumerà in sé la parte che gli compete dell'oggetto PRECEDENTE: per QUESTA RAGIONE quello deve essere stato creato prima! Tale parte viene accolta in forma privata, ma la variabile *sesso* e il tipo *essere\_umano::Sesso* sono di nuovo esposti a un'eventuale ulteriore erede, grazie all'uso qui compiuto della parola di vocabolario *using*;
3. realizzazione di un oggetto della classe *studente\_di\_fisica*, che assume in sé la parte che gli compete dell'oggetto PRECEDENTE, il quale ha generosamente provveduto a rendere disponibile, ex punto precedente, PURE quanto ricevuto da *essere\_umano*, e assume anche il nome di *amilcare*, valido per tutto l'ambito di *main*. Tale nome è il solo *stargate*, a meno di eventuali puntatori a membro, che prima o poi si incontreranno, di cui *main* dispone per poter accedere alle informazioni esclusive detenute internamente dall'oggetto, comprese quelle pertinenti agli oggetti innominati delle classi antenate, che ne fanno parte.

Siccome la Coerenza è una delle virtù del buon programmatore, le affermazioni 1. e 2. del precedente elenco IMPLICANO che siano eseguiti, nello stesso ordine, e PRIMA del costruttore già individuato per *studente\_di\_fisica*, UN costruttore di *essere\_umano* e UNO di *studente*. Nel caso presente non ci sono possibilità di scelta, visto che OGNI classe dispone di un UNICO costruttore; cionondimeno va osservato che, quantunque NON POSSA SCEGLIERE, è proprio il costruttore di *studente\_di\_fisica*, pur eseguito per ULTIMO, a fornire il materiale da costruzione al costruttore di *studente*; e questi lo fornisce, a sua volta, al costruttore di *essere\_umano*, di modo che il cemento e i mattoni preparati da *main* arrivano fino alla cima della piramide edificativa (GRANDISSIMO LINGUAGGIO). Se poi il costruttore di *studente\_di\_fisica* avesse anche la possibilità di SCEGLIERE tra diversi costruttori di *studente*, e così via, a ritroso, lascio a VOI immaginare il numero di opportunità diverse che si potrebbero aprire, e che SI APRIRANNO: AFASÌA....  
  
Non dimenticate, però, che è stato detto che l'invocazione di un costruttore della classe antecedente, che può avvenire inserendola nella lista degli inizializzatori di un costruttore di una classe erede, È SOLO OPZIONALE: nel caso presente, se non venisse fatta, occorrerebbe equipaggiare le classi antecedenti ANCHE con un costruttore di default, perché allora sarebbe QUELLO a essere eseguito per realizzare *amilcare*, e il *costruttore di default* non è più fornito *gratis et amore Dei* dal compilatore, perché almeno UN costruttore è stato definito esplicitamente in ciascuna classe (ricordate? **RICORDATE!**).  
  
Quando il costruttore di *essere\_umano* inizia a eseguirsi, E SOLO ALLORA, prende dalla memoria del calcolatore TUTTO QUELLO che serve a far esistere un oggetto *essere\_umano*, vale a dire lo spazio di memoria in cui collocare OGNI variabile membro di quella classe, e che resterà occupato per TUTTA la durata della vita dell'oggetto: il suo ambito, lo avete presente? In altri termini la memoria per gli oggetti è catturata dai costruttori, come è giusto che sia, visto anche che sono eseguiti AL MOMENTO DELLA DICHIARAZIONE, e **NON AL COMPLETAMENTO DELLA DEFINIZIONE DELLA CLASSE** la quale, di suo, non occupa un'emerita cippa, quand'anche vi fosse dichiarato come membro un *array* da 100 *terabytes*: **la definizione di una classe è UN PROGETTO, non UN OGGETTO**.  
  
All'affermazione appena formulata, come ennesimo ASSIOMA, ma neanche tanto, sfuggono i membri di una classe qualificati con la parola di vocabolario *static*, di cui la variabile *size\_t studente\_di\_fisica :: massima\_lunghezza* è un esempio. Per essi la memoria viene presa dal *linker* prima ancora che il programma cominci a essere eseguito, ed è PER QUESTA RAGIONE che DEVONO ESSERE RIDICHIARATI nell'ambito globale dato che il *linker* può prendere memoria fin dall'inizio SOLO PER CIÒ CHE SI TROVA LÌ e che è destinato a permanere per TUTTA la durata dell'esecuzione.   
  
Ora **accendete i neuroni** sull'ultima proposizione e traetene le conseguenze LOGICHE: se un membro qualificato *static* vien fatto esistere in memoria fin dall'istante immediatamente successivo al *big bang*, quando ancora non esiste alcun oggetto della sua classe, e PERMANE per TUTTA la durata dell'esecuzione del programma, sopravvivendo a qualunque oggetto della sua classe che nasca e muoia durante l'esecuzione stessa...tutto questo non può che implicare quanto segue:

1. ogni membro qualificato *static* è UNICO nella memoria, indipendentemente da quanti oggetti della sua classe saranno eventualmente realizzati e/o distrutti;
2. a un membro qualificato *static* accedono, con ogni diritto, TUTTI gli oggetti della sua classe, anche quelli futuri;
3. se un membro qualificato *static* è dichiarato nella zona *public:* della classe, a quel membro può accedere, con ogni diritto, qualsiasi parte del programma successiva alla sua dichiarazione globale, in qualsiasi momento dell'esecuzione e INDIPENDENTEMENTE DAL FATTO che ci siano, o no, oggetti di quella classe disponibili;
4. per quanto appena detto, a un membro pubblico qualificato *static* deve potersi accedere ANCHE *"a prescindere"*, come direbbe Totò, da un oggetto e dall'uso dell'operatore punto: ciò si ottiene semplicemente citandolo col suo nome, fatto precedere dal risolutore di ambito. Ad esempio, alla variabile  
   *studente\_di\_fisica :: massima\_lunghezza*,  
   essendo appunto pubblica, si può accedere ESATTAMENTE come si è APPENA SCRITTO.

Quanto scritto nell'elenco puntato spiega come mai il costruttore di ogni *studente\_di\_fisica* possa confrontare la lunghezza del nome che gli viene trasmesso, restituita dalla funzione *strlen*, col valore di *massima\_lunghezza*, aggiornandolo, se occorre, in vista di un analogo controllo compiuto da costruttori futuri, grazie alla funzione *max* il cui significato dovrebbe essere intuitivo e la cui spiegazione dettagliata è prematura; esiste anche una *min*.  
  
Proseguendo la lettura di *main*, dopo la realizzazione di *amilcare* avvengono esattamente le stesse cose per la realizzazione di *teodolinda*, salvo il fatto che al costruttore viene trasmesso anche il secondo argomento per non fargli usare il valore *standard*, come era avvenuto per *amilcare*. La scelta del valore *standard* del secondo argomento del costruttore è la sola asimmetria sessuale che l'autore, essendo un *vir*, si è concessa: tuttavia si potrebbe tranquillamente rinunciare a porre come *standard* il secondo argomento e fornirlo sempre e comunque, ripristinando in tal modo una simmetria completa tra i due sessi.  
  
La forma della trasmissione del secondo argomento al costruttore di *studente\_di\_fisica* per l'istanziazione di *teodolinda* consente di spendere qualche parola sul suo tipo, che, come si vede, va scritto come *essere\_umano :: Sesso*, lo stesso della variabile *sesso* dell'ambito di *main*, che così viene coerentemente inizializzata a volo col successivo *:: femmina*, che ne dà il valore.  
  
L'apparentemente complicata sintassi è spiegata in poche parole: il risolutore di ambito *essere\_umano ::* è necessario perchè il tipo *Sesso* è stato definito nell'ambito di quella classe, NON nell'ambito di *main* o in quello globale; peraltro *main* LO PUÒ USARE perché la definizione ha avuto luogo NELLA ZONA *public:* di *essere\_umano*. Vedete perché la COERENZA è una virtù indispensabile? Provate a sostituire *public* con *protected* e ve ne accorgerete....  
  
*Sesso* è, appunto, il nome attribuito al tipo; e risulta definito come una *enum class* con *bool* come tipo sottostante. Potrete approfondire a tempo debito questi concetti andando a visitare la pagina dedicata a *enum*, ma per adesso vi basti sapere che *bool* è semplicemente il tipo attribuito agli enumeratori, ossia alle costanti contenute nella coppia di graffe: basta vedere come sono inizializzate; tali costanti, in una *enum class*, diversamente da quanto avviene in una semplice *enum*, che abbiamo già incontrato (ricordate la *enum settimana*? Ma non ricordate **PROPRIO NIENTE**?), sono circoscritte al loro ambito, e questo spiega come mai, per potervi accedere, occorra l'ulteriore risolutore *Sesso ::*. Va notato (NOTATELO!) che, all'interno delle classi *studente* e *studente\_di\_fisica*, lo stesso tipo può essere usato SENZA il risolutore *essere\_umano ::* , dato che entrambe da questa lo ereditano.  
  
Realizzata anche *teodolinda*, con tutta la memoria che le serve, DIVERSA E SEPARATA da quella di *amilcare*, salva la condivisione della cella PERMANENTE occupata dall'UNICO membro *static*, come detto sopra, il programma si lancia alla realizzazione addirittura di un *array* di *studente\_di\_fisica*, chiamato appunto *array\_di\_studenti*, il cui contenuto è immediatamente inizializzato grazie alla presenza della coppia di graffe contenente gli inizializzatori separati con DUE virgole, né più né meno di come potrebbe avvenire per un semplice *array* di interi.  
  
Il fatto che ci siano due virgole IMPLICA che gli inizializzatori sono TRE e tale sarà pertanto il numero di oggetti contenuti nell'*array*, opportunamente omesso entro le parentesi quadre. Ma osservate la forma dei TRE inizializzatori: si tratta di TRE INVOCAZIONI **ESPLICITE** dello stesso costruttore di *studente\_di\_fisica* che ha già costruito anche *amilcare* e *teodolinda*, essendo, in quei casi, invocato **IMPLICITAMENTE**: che LAVORI, visto che è l'unico costruttore che c'è nella classe!  
  
La morale è che un costruttore può anche essere invocato esplicitamente, come si vede, ma l'utilità di una simile invocazione è ristretta a pochi contesti, ossia a quelli in cui l'oggetto che il costruttore ESPLICITAMENTE INVOCATO costruisce, perché un costruttore sa fare solo questo..., NON HA BISOGNO DI UN NOME.  
Nel contesto attuale, a ben vedere, accade proprio così, dato che il nome dei tre oggetti costruiti è già insito nel nome dell'*array* che li contiene, e nell'ambito di *main*, si chiameranno infatti, rispettivamente, *array\_di\_studenti[0]*, *array\_di\_studenti[1]* e *array\_di\_studenti[2]*, come è ampiamente noto.  
  
Altri contesti in cui può essere utile invocare esplicitamente un costruttore sono quelli in cui occorra introdurre un cosiddetto riferimento destro a un oggetto di una certa classe ...*"ma non eran da ciò le proprie penne"*, direbbe il Poeta; almeno per adesso, aggiungo io.  
  
Dopo aver costruito tutti gli oggetti *studente\_di\_fisica*, cinque in tutto, e se volete capacitarvi che le cose vadano davvero come vi ho detto, provate ad aggiungere nell'ambito dei vari costruttori una semplice espressione *cout* che scriva un messaggio riconoscibile, e vedrete quante volte, e in quale ordine, lo leggerete, *main* non fa altro che scriverli tutti sullo *standard output stream*, dimostrando di aver capito ogni cosa a dovere.  
  
Va osservato, però, COME li scrive: esattamente come sarebbe scritta una variabile di qualsiasi tipo nativo.  
  
Ricordate quando, nei canti 20 e 21, si parlava di funzioni sottintese che venivano eseguite quando si incontrava l'operatore di inserimento (*<<*) con a sinistra un oggetto gestore di *output stream*? Qui smettono di essere sottintese e appaiono in tutta la loro gloria. Ce ne sono due, sintatticamente identiche, una che viene eseguita quando alla destra di *<<* appare un oggetto della classe *studente\_di\_fisica* e l'altra quando vi appare un oggetto di tipo *essere\_unano::Sesso*: il compilatore capisce quale far eseguire semplicemente esaminando quale tipo del secondo argomento che compare nei due *overload* scritti nel codice combaci con quello dell'operando destro di *<<*. Ne consegue, con tutta evidenza, che è possibile scrivere una funzione rispondente allo stesso modello PER OGNI CLASSE tra le infinite che il buon programmatore possa immaginare, e, in verità, anche più di una sola per classe.  
  
Esaminiamo da vicino queste funzioni: entrambe ricevono come primo argomento un riferimento sinistro a un oggetto della classe *ostream* che, al termine della loro esecuzione, restituiscono a chi le ha eseguite: la classe *ostream* è antenata di tutte le classi cui appartengono gli oggetti gestori di OGNI genere di *output stream*, proprio come la nostra *essere\_umano* è antenata di *studente\_di\_fisica*; e quindi, come si può affermare che ogni *studente\_di\_fisica* è anche, e prima di tutto, un *essere\_umano*, così si potrebbe dire che tanto *ofstream* quanto *ostringstream* sono anche, e PRIMA, delle *ostream*. L'analogia non è rigorosamente perfetta, perché a *essere\_umano* fa difetto una caratteristica importante di *ostream* che qui non è ancora il caso di specificare, ma è sufficiente, forse, a far capire come mai vada preferito *ostream* come tipo del primo argomento, e quindi come tipo restituito, così da essere valido e comune per tutti i tipi di oggetti da gestire in un programma normale.  
  
La restituzione dello stesso riferimento ricevuto come argomento è garanzia per la buona riuscita della ricorsività delle operazioni di inserimento nell'*output stream*; non è che sia vietato far restituire qualsiasi altro tipo, o anche nulla, dichiarando la funzione di tipo *void*, finché si mantiene il tipo giusto per l'argomento trasmesso: basta che si sia consapevoli che questo comporterebbe la necessità di non scrivere espressioni di inserimento in *output* che contengano più di un'UNICA VOLTA l'operatore *<<*, e questa, a ben pensarci, non è poi una così buona idea.  
  
Se poi uno vuol essere masochista fino in fondo può anche decidere di trasmettere come argomento, e farsi restituire, un riferimento alla classe *ofstream*, tanto per dirne una, piuttosto che a *ostream*...basta che sia COERENTE e rinunci non solo alla piena ricorsività dell'operatore, ma anche a usarlo con *output stream* diversi da quelli della classe specificamente utilizzata.  
  
Le seguenti due varianti dello STESSO programma riportato sopra illustrano quanto detto; nella prima si fa restituire un intero a una delle due funzioni *operator<<*, nella seconda si trasmette a entrambe appunto un riferimento a *ofstream* e ci si regola con coerenza in entrambe le varianti. Trovate e spiegate a voi stessi TUTTE LE DIFFERENZE, perché TUTTE sono irrinunciabili se non si vuole rinunciare a far funzionare bene il programma:  
  
**prima variante**, con restituzione di un *int*, o di che altro si voglia

# include <iostream>  
# include <iomanip>  
# include <string>  
# include <cstring>  
using namespace std;  
  
class essere\_umano  
{public:  
enum class Sesso : bool  
{maschio=true, femmina=false};  
protected:  
Sesso sesso;  
essere\_umano(Sesso b) {sesso = b;}};  
  
class studente : essere\_umano  
{protected:  
string nome;  
using essere\_umano :: sesso;  
public:  
using essere\_umano :: Sesso;  
studente(const char \* s, Sesso b) :  
nome(s), essere\_umano(b) { }};  
  
class studente\_di\_fisica : studente  
{friend int operator <<  
(ostream &, studente\_di\_fisica);  
public:  
static size\_t massima\_lunghezza;  
Sesso rendiSesso( ) {return sesso;}  
studente\_di\_fisica  
(const char \*s, Sesso b = Sesso :: maschio) :  
studente(s, b)  
{massima\_lunghezza =  
max(massima\_lunghezza, strlen(s));}};  
  
size\_t studente\_di\_fisica :: massima\_lunghezza = 0;  
  
ostream & operator <<  
(ostream &o, essere\_umano :: Sesso s)  
{string sessi[ ]  
{"femminile", "maschile"};  
bool  
indice =  
static\_cast<bool>(s) ==   
static\_cast<bool>(essere\_umano :: Sesso :: maschio)  
&& sessi[true] == "maschile"  
||  
static\_cast<bool>(s) ==   
static\_cast<bool>(essere\_umano :: Sesso :: femmina)  
&& sessi[true] == "femminile";  
return o << sessi[indice];}  
  
int operator <<  
(ostream &o, studente\_di\_fisica s)  
{o . setf(ios :: left, ios :: adjustfield);  
o << setw(1+s.massima\_lunghezza)  
<< s . nome << " di sesso " << s . sesso;  
return 1;}  
  
int main( )  
{essere\_umano :: Sesso sesso;  
int quanti = 0;  
studente\_di\_fisica  
amilcare("Amilcar de Chartaginensibus"),  
teodolinda  
("Teodolinda de Bavaris",  
sesso = essere\_umano :: Sesso :: femmina),  
array\_di\_studenti[ ]  
{studente\_di\_fisica  
("Federico Barbarossa",  
sesso = essere\_umano :: Sesso :: maschio),  
studente\_di\_fisica  
("Pia de' Tolomei",  
sesso = essere\_umano :: Sesso :: femmina),  
studente\_di\_fisica  
("Gerolamo Savonarola")};  
clog   
<< "ecco gli oggetti \"studente\_di\_fisica\"\n";  
quanti += cout << amilcare; cout << '\n';  
quanti += cout << teodolinda; cout << '\n';  
for(studente\_di\_fisica stud : array\_di\_studenti)  
{quanti += cout << stud; cout << '\n';}  
cout << "scritti " << quanti <<  
" \"studenti\_di\_fisica\"\n";}

**seconda variante**, con trasferimento di *ofstream*

# include <iostream>  
# include <fstream>  
# include <iomanip>  
# include <string>  
# include <cstring>  
using namespace std;  
  
class essere\_umano  
{public:  
enum class Sesso : bool  
{maschio=true, femmina=false};  
protected:  
Sesso sesso;  
essere\_umano(Sesso b) {sesso = b;}};  
  
class studente : essere\_umano  
{protected:  
string nome;  
using essere\_umano :: sesso;  
public:  
using essere\_umano :: Sesso;  
studente(const char \* s, Sesso b) :  
nome(s), essere\_umano(b) { }};  
  
class studente\_di\_fisica : studente  
{friend ofstream & operator <<  
(ofstream &, studente\_di\_fisica);  
public:  
static size\_t massima\_lunghezza;  
Sesso rendiSesso( ) {return sesso;}  
studente\_di\_fisica  
(const char \*s, Sesso b = Sesso :: maschio) :  
studente(s, b)  
{massima\_lunghezza =  
max(massima\_lunghezza, strlen(s));}};  
  
size\_t studente\_di\_fisica :: massima\_lunghezza = 0;  
  
ofstream & operator <<  
(ofstream &o, essere\_umano :: Sesso s)  
{string sessi[ ]  
{"femminile", "maschile"};  
bool  
indice =  
static\_cast<bool>(s) ==   
static\_cast<bool>(essere\_umano :: Sesso :: maschio)  
&& sessi[true] == "maschile"  
||  
static\_cast<bool>(s) ==   
static\_cast<bool>(essere\_umano :: Sesso :: femmina)  
&& sessi[true] == "femminile";  
o << sessi[indice];  
return o;}  
  
ofstream & operator <<  
(ofstream &o, studente\_di\_fisica s)  
{o . setf(ios :: left, ios :: adjustfield);  
o << setw(1+s.massima\_lunghezza)  
<< s . nome << " di sesso ";  
return o << s . sesso;}  
  
int main( )  
{essere\_umano :: Sesso sesso;  
ofstream os("OS");  
studente\_di\_fisica  
amilcare("Amilcar de Chartaginensibus"),  
teodolinda  
("Teodolinda de Bavaris",  
sesso = essere\_umano :: Sesso :: femmina),  
array\_di\_studenti[ ]  
{studente\_di\_fisica  
("Federico Barbarossa",  
sesso = essere\_umano :: Sesso :: maschio),  
studente\_di\_fisica  
("Pia de' Tolomei",  
sesso = essere\_umano :: Sesso :: femmina),  
studente\_di\_fisica  
("Gerolamo Savonarola")};  
clog   
<< "ecco gli oggetti \"studente\_di\_fisica\"\n";  
os << amilcare << '\n';  
os << teodolinda << '\n';  
for(studente\_di\_fisica stud : array\_di\_studenti)  
os << stud << '\n';}

Non si ritiene di dover proporre una terza variante che cumuli le due riportate. Piuttosto mette conto osservare come il programma usi, ancora una volta, il manipolatore *setw*, in cooperazione col valore della variabile *massima\_lunghezza*, per scrivere i dati in forma tabulare e come il metodo *setf*, che troverete descritto nella pagina di dettaglio sulle scritture formattate, ottenga la giustificazione a sinistra delle stringhe al posto di quella normale a destra.  
  
E più di ogni altra cosa occorre notare la dichiarazione come *friend*, nella classe *studente\_di\_fisica*, del prototipo della sua *operator<<*: con tale dichiarazione si informa il compilatore che, a un'eventuale funzione che avesse QUELLA SEGNATURA, sono concessi tutti i diritti di accesso ai membri della classe, ANCHE A QUELLI PRIVATI come *nome*, che infatti viene usato impunemente nell'ambito della funzione. Si noti: EVENTUALE significa che poi non si è obbligati a scriverla per forza, se non la si usasse. L'altra *operator<<*, quella riferita a *essere\_umano::Sesso*, non abbisogna di essere dichiarata *friend* di chicchessia, perché il tipo a cui si riferisce è di pubblico dominio, una volta individuato col suo risolutore di ambito.

**Canto ventiseiesimo: le funzioni *operator*, il puntatore *this* e il costruttore con un solo argomento**

*Ma sì com'elli avvien, s'un cibo sazia  
e d'un altro rimane ancor la gola,  
che quel si chere e di quel si ringrazia,  
  
così fec'io con atto e con parola,  
per apprender da lei qual fu la tela  
onde non trasse infino a co la spola.*   
  
Se, dopo essere stati saziati del cibo delle spiegazioni di come avvenga che l'operatore *<<* sia trasformato sì da poter far inserire checchessia in un *output stream*, non vi è rimasta la gola di sapere se sia possibile trarre ancor più avanti la spola di questa tela...  
...meglio che vi orientiate al più presto ad altre attività.   
  
Per i sufficientemente golosi di conoscenza dirò subito, senza frapporre alcun ulteriore indugio, che la parola di vocabolario *operator* consente di fare molto, ma molto, di più, ossia di RIDEFINIRE **TUTTI GLI OPERATORI UNARI E BINARI** PREVISTI DAL LINGUAGGIO (con le sole eccezioni, peraltro comprensibilissime, dell'operatore punto, del suo compare *.\** e del risolutore di ambito *::*) in relazione a COME DEBBANO OPERARE quando ALMENO UN OPERANDO appartiene a una classe definita dal programmatore.  
  
**È ESATTAMENTE QUEL CHE È STATO FATTO NEL CANTO PRECEDENTE PER L'OPERATORE *<<***, usato come precursore di quanto si vedrà nel canto presente.  
  
Sgombriamo subito il campo da possibili equivoci: la ridefinizione di cui si parla NON ALTERA le caratteristiche dell'operatore per quanto concerne la precedenza e l'associatività, che permangono quali gli competono e quali voi dovreste già conoscere; altrimenti **RIGUARDÀTEVELE**. **TANTO MENO CONSENTE** di creare operatori che non esistono.  
  
Assodato ciò per sempre, la sintassi da usare è proprio quella che si è già vista; se, come ulteriore esempio, si volesse ridefinire l'operatore di addizione tra una variabile intera e uno *studente\_di\_fisica* (non si sa perché, ma tanto è solo un esempio...magari per aggiungere un voto al *curriculum*...) si potrebbe definire la seguente funzione:

int operator + (int n, studente\_di\_fisica ciccio)  
{  
int intero\_da\_restituire;  
//qualche oscura operazione che inizializzi  
//*intero\_da\_restituire* in espressioni che  
//coinvolgano i due argomenti ricevuti  
return intero\_da\_restituire;}

**Osservazioni:**

* questa funzione verrà AUTOMATICAMENTE ESEGUITA per determinare il valore (che sarà pari a quello restituito) di qualsiasi sottoespressione in cui l'operatore binario *+* abbia un *int* come operando sinistro e uno *studente\_di\_fisica* come operando destro. Naturalmente una tale sottoespressione può comparire nel codice del programma SOLO DOPO la definizione di questa funzione;
* il valore dei due operandi viene ricevuto dalla funzione nei suoi due argomenti. L'ordine degli argomenti rispecchia quello degli operandi; ne segue che la funzione qui proposta NON VIENE INVOCATA per risolvere una sottoespressione in cui *studente\_di\_fisica* sia l'operando sinistro e l'intero sia quello destro: una sottoespressione così ordinata CAGIONA ERRORE, almeno fino a quando non viene definita una seconda funzione *operator+* in *overload*, con la lista degli argomenti invertita.
* non ha alcuna rilevanza il numero di spazi (anche nessuno) eventualmente frapposti fra la parola di vocabolario *operator* e il segno grafico dell'operatore interessato: qualunque sia tale spaziatura nella definizione portata come esempio, la crasi (anche con spazi frapposti) *operator+* finisce per essere equiparata al nome della funzione, al punto che questa potrebbe perfino essere invocata, con la stessa sintassi di ogni altra funzione, come nella seguente espressione (in cui i parametri trasmessi sono supposti adeguatamente dichiarati e inizializzati):  
  *cout << operator+(un\_certo\_intero, un\_certo\_studente) << '\n';*
* il tipo del risultato, ossia ciò che la funzione restituisce al chiamante, è **A TOTALE CARICO E DISCREZIONE** del programmatore; secondo la scelta che egli/ella farà, tuttavia, assumeranno o no validità determinate espressioni, tenuto conto del fatto che precedenza e associatività dell'operatore coinvolto restano, come detto, INTATTE. Fermo restando l'esempio, e assumendo dichiarati e inizializzati in modo acconcio gli oggetti *intero* e *studentello*, ecco una lista (NON esaustiva) di dichiarazioni inizializzate con a fianco un commento sulla loro validità. Spiegatevi DA SOLI il perché; dovreste esserne capaci. Se non ci riuscite, domandate di persona:
  + *int n = intero + intero + studentello; // VALIDA*
  + *int n = intero + studentello + intero; // VALIDA*
  + *int n = studentello + intero + intero; // NON VALIDA*
  + *int n = intero + studentello + studentello; // VALIDA*
  + *int n = intero \* intero + studentello; // VALIDA*
  + *int n = studentello + intero \* intero; // NON VALIDA*
  + *int n = intero + intero \* studentello; // NON VALIDA*
  + *int n = intero \* (intero + studentello); // VALIDA*
  + *// e via dicendo ...*
* nel momento in cui si opera la definizione della funzione *operator* la classe, o le classi, cui appartiene o appartengono gli argomenti deve, o devono, evidentemente essere già stata dichiarata, o dichiarate, ma anche completata, o completate. La completezza è naturalmente richiesta nel momento in cui si pretenda che la funzione sia eseguita;
* se la funzione *operator* pretende/necessita accesso ai membri non pubblici dei suoi argomenti, allora nella definizione della classe di pertinenza DEVE ESSERE INSERITA una **dichiarazione** (ho scritto dichiarazione, **NON** definizione...) della funzione *operator* stessa, preceduta dalla parola di vocabolario *friend*. Tale qualifica NON SI ESTENDE ad eventuali classi eredi, nella cui definizione, qualora occorresse, dovrebbe essere esplicitamente ripetuta;
* qualunque funzione, su insindacabile decisione del programmatore, può essere qualificata *friend* di una certa classe, applicando per essa lo stesso procedimento descritto al punto precedente, anche se non riceve come argomenti oggetti di quella classe. In tal caso è chiaro che tale qualifica assume efficacia SOLO se la funzione realizza, nel proprio ambito, oggetti della classe di cui è *friend*: relativamente a questi, avrebbe gli stessi diritti di accesso dei metodi della classe cui essi appartengono;
* che non vi salti in mente di fare gli *snob* o i *"nerd per forza"* provando a pretendere di ridefinire come vi garba l'azione degli operatori sui tipi nativi, scrivendo qualcosa come:

int operator+(int x, int y)   
{int ris; /\*omissis\*/ return ris;}

Il compilatore vi riderebbe in faccia e vi umilierebbe ricordandovi quanto scritto nel secondo capoverso di questo stesso canto.

Una funzione *operator* può anche essere membro di una classe, anzi questa pratica è quella maggiormente utilizzata nella redazione dei programmi C++. In tal caso, però, il numero di argomenti che la funzione riceve diminuisce sistematicamente di un'unità, in quanto uno degli operandi, o il solo operando, si assume tacitamente essere l'oggetto che detiene la funzione; nel caso degli operatori binari tale operando tacitamente identificato è SEMPRE l'operando **SINISTRO**.  
  
Un solo esempio varrà verosimilmente più di molte parole:

#include <iostream>  
#include <cmath>   
using namespace std;  
  
class Double  
{bool errore;  
double d;  
public:  
  
// addizione (e altre op.) Double+double  
Double operator+(double s)  
{Double r = d; r.d += s; return r;}  
// definire anche, analogamente,  
// operator- e operator\*  
  
// invece operator/ è definita:  
Double operator/(double s)  
{Double r = d;  
r.errore = s == 0.0;  
if(!r.errore) r.d/=s;  
return r;}  
  
  
// operatori - e + UNARI  
Double operator-( )  
{d \*= -1.0; return \*this;}  
Double operator+( )  
{return \*this;}  
  
  
// addizione (e altre op.) Double+Double  
Double operator+(Double d)  
{Double r = this->d; r.d += d.d; return r;}  
// definire anche, analogamente,  
// operator- e operator\*  
  
// invece operator/ è definita:  
Double operator/(Double d)  
{Double r = this->d;  
r.errore = d.d == 0.0;  
if(!r.errore) r.d /= d.d;  
return r;}  
  
  
// addizione (e altre op.) double+Double  
friend Double operator+(double, Double);  
// aggiungere le altre dichiarazioni  
  
  
// addizione con assegnamento Double += double  
Double operator+=(double s)  
{d+=s; return \*this;}  
// definire anche, analogamente,  
// operator-= e operator\*=  
  
// invece operator/= è definita:  
Double operator/=(double s)  
{errore = s == 0.0;  
if(!errore) d/=s;  
return \*this;}  
  
  
// addizione con assegnamento Double += Double  
Double operator+=(Double s)  
{d += s.d; return \*this;}  
// definire anche, analogamente,  
// operator-= e operator\*=  
  
// invece operator/= è definita:  
Double operator/=(Double s)  
{errore = s.d == 0.0;  
if(!errore) d/=s.d;  
return \*this;}  
  
  
// addizione con assegnamento double += Double  
friend double& operator+=(double&, Double);  
// aggiungere le altre dichiarazioni  
  
  
// definizione del calcolo percentuale Double % double  
double operator%(double d)  
{return d \* this->d / 100.0;}  
  
// definizione del calcolo percentuale Double % Double  
double operator%(Double d)  
{return d.d \* this->d / 100.0;}  
  
// definizione del calcolo percentuale double % Double  
friend double operator%(double, Double);  
  
  
// output stream  
friend ostream& operator<<(ostream &, Double);  
  
  
// funzioni matematiche  
friend Double exp(Double);  
// dichiarare altre funzioni che interessino  
  
  
// costruttore  
Double(double s) : d(s) {errore = false;}  
  
// restituzione di errore (con annessa falsificazione)  
bool Errato( )  
{if(errore) return !(errore=false);  
return errore;}  
  
  
// restituzione diretta della variabile membro privata  
double rendi( )  
{return d;}  
  
// casting da Double a double  
operator double( )  
{return d;}  
}; // fine della definizione della classe (completata)  
  
// definizione delle funzioni friend (FUORI dalla classe!)  
  
Double operator+(double d, Double D)  
{Double r = d; r.d += D.d; return r;}  
// definire le altre  
  
double& operator+=(double &d, Double D)  
{d += D.d; return d;}  
// definire le altre  
  
double operator % (double d, Double D)  
{return D.d \* d / 100.0;}  
  
ostream& operator<<(ostream &o, Double D)  
{return o << D.d;}  
  
Double exp(Double x = 1.0)  
{Double r = 1.0; r.d = exp(x.d); return r;}  
  
  
int main( )  
{  
double (Double::\*d)(  ) = &Double::rendi;  
double tasso = 12.8;  
Double quota = 34567.789;  
cout << "posso usare il puntatore a funzione membro \"d\""  
"\nper eseguire la funzione rendi e ottenere il valore della"  
"\nvariabile privata di un oggetto Double che risulta "  
<< (quota .\* d)(  ) << '\n';  
cout  
<< "posso eseguire double + Double: "  
<< tasso + quota << '\n';  
cout  
<< "ma anche Double + double: "  
<< quota + tasso << '\n';  
cout  
<< "e financo Double + Double: "  
<< quota + Double(tasso) << '\n';  
cout  
<< "poi posso calcolare il "  
<< tasso << "% di " << quota  
<< "\nsia come Double % double: "  
<< quota % tasso << '\n';  
cout  
<< "sia come double % Double: "  
<< (double)quota % Double(tasso) << '\n';  
cout  
<< "sia come Double % Double: "  
<< quota % Double(tasso) << '\n';  
cout  
<< "Posso constatare se funziona la divisione\n";  
if((quota / 0.0) . Errato( ))  
cerr << "PIRLA! non si divide per 0\n";  
if(!(quota /= 2.0) . Errato( ))  
cout  
<< "per 2 invece puoi: "  
<< quota << '\n';  
cout  
<< "Posso anche incrementare "  
<< quota << " di " << tasso;  
cout  
<< " ottenendo " << (quota += tasso) << '\n';  
cout  
<< "e anche incrementare " << tasso  
<< " del valore incrementato di " << quota  
<< " ottenendo "; cout << (tasso += quota) << '\n';  
cout  
<< "posso applicare il - unario a "  
<< quota << " ottenendo ";  
cout << -quota << '\n';  
cout  
<< "e incrementare con questo il valore attuale di "  
<< tasso   
<< "\nlavorando con operator+= tra due Double, e ottenendo ";  
cout  
<< (tasso = (double)(Double(tasso) += quota)) << '\n';  
cout  
<< "infine posso calcolare l'esponenziale di un Double come "  
<< Double(tasso) << " ottenendo "  
<< exp(Double(tasso)) << '\n'  
<< "e anche scrivere il valore del numero di Nepero che è "  
<< exp( ) << '\n';  
}

**ESEGUITE IMMANTINENTE** questo programma, perché sarà ricchissimo di insegnamenti per voi: la classe *Double* che propone, una volta che fossero completate le definizioni delle funzioni suggerite e lasciate, per brevità, solo allo stato di commento, realizza un tipo che ha la medesima funzionalità del tipo nativo *double* con, in aggiunta, la possibilità di utilizzare anche, per il calcolo della percentuale, l'operatore *%* che si ricorderà, spero, essere vietato tra variabili *double*; così, tanto per fare un esempio...poi il fatto che l'operatore *%* restituisca un *double* piuttosto che un *Double* è una scelta come un'altra.  
Inoltre la classe *Double* è perfettamente interfacciata anche col tipo nativo, consentendo tutte le operazioni aritmetiche miste tra *Double* e *double* in qualunque ordine, oltre che quelle pure tra due operandi entrambi *Double*, e perfino il *casting* nei due sensi.  
Qui appresso si dà un elenco degli insegnamenti che dovreste trarre da questo programma:

1. le qualifiche *friend* riguardano SOLO, ovviamente, le funzioni che NON siano membri della classe: tra esse, oltre alle *operator* esterne, sono state inserite anche le *overload* delle funzioni matematiche elementari, in modo che possano essere eseguite come d'abitudine e NON, come pure sarebbe consentito, come membri, per i quali sarebbe stata necessaria la sintassi che fa uso dell'operatore . (punto);
2. in TUTTI i membri *operator* per operatori binari, come si può facilmente osservare, l'oggetto *Double* proprietario del metodo rappresenta SEMPRE l'operando **SINISTRO**: quello destro entra nella funzione come il suo unico argomento;
3. nei due membri *operator* per operatori unari l'UNICO operando è l'oggetto detentore stesso: infatti la lista degli argomenti è vuota. Si osservi che il metodo *operator-* unario MODIFICA il proprio operando, ossia il proprio oggetto proprietario: si tratta di una scelta non necessaria operata dal programmatore. Si osservi anche come, nell'ambito di *operator-( )*, così come nell'ambito di qualsiasi altro metodo membro, la variabile *d* risulti già dichiarata: questo NON è sorprendente perché conferma la regola degli ambiti esterni visibili anche in quelli interni. Tuttavia va osservato che lo stesso accadrebbe ANCHE se un metodo membro fosse definito all'esterno dell'ambito della classe: in altre parole l'ambito della classe è considerato sempre inclusivo rispetto agli ambiti dei propri metodi;
4. OGNI FUNZIONE MEMBRO di una classe, che **non sia stata qualificata come *static***, dispone gratuitamente, nel proprio ambito, di un puntatore all'oggetto di cui fa parte, cui è possibile riferirsi tramite la parola di vocabolario ***this***; tale puntatore è stato utilizzato, nel corso del programma, in diversi contesti:
   * nei membri *operator* per gli operatori aritmetici con assegnamento, al fine di restituire il valore dell'oggetto stesso che ne costituiva l'operando SINISTRO, che risulta anche, come effetto collaterale, coerentemente modificato; tale scelta consente, ad esempio, di ottenere la scrittura del valore aggiornato di *quota* nell'espressione *cout << (quota += tasso).* Non dovrebbe occorrere che vi venga ripetuto che le parentesi, in questa espressione, sono NECESSARIE;
   * nei membri *operator* per gli operatori aritmetici unari, per restituire il valore dell'oggetto, INTATTO (*operator+( )*) o MODIFICATO (*operator-( )*);
   * nei membri che volutamente, per creare un caso didascalico, ricevono un argomento omonimo della variabile membro *d*, per distinguerne quest'ultima;
5. le funzioni *friend* NON POSSONO citare il puntatore *this*, semplicemente perché NON FANNO PARTE DELLA CLASSE;
6. neppure possono citarlo i metodi membri che siano qualificati *static*, non perché non facciano parte della classe (NE FANNO PARTE), ma perché, al pari delle variabili membro ugualmente qualificate, sussistono a prescindere dall'esistenza di qualunque oggetto e pertanto, nel loro ambito, il puntatore *this* rischierebbe concretamente di puntare a vuoto ed è ovvio che il compilatore NON POSSA ESSERE PROPRIO LUI A CONSENTIRLO;
7. i più acuti tra voi avranno già autonomamente concluso che, in base a quanto appena detto, la LOGICA impone che un metodo membro che sia qualificato *static* possa accedere, nel proprio ambito, SOLO ad altri membri a loro volta così qualificati...BRAVI! Avete indovinato! È, in effetti, esattamente così;
8. ...e verrebbe da chiedersi, se siete vivi, *"perché qualificare static un metodo membro?"*;
9. e a me verrebbe istintivo rispondervi che movete tenerezza qual *"un fante/ che bagni ancor la lingua a la mammella"*; per non privarvi del latte della conoscenza risponderò con una domanda che cela in sé la risposta: come altrimenti potreste gestire una variabile membro qualificata *static* che non fosse pubblica?
10. nella funzione *friend* che ridefinisce l'operatore *+=* quando i suoi operandi sono, nell'ordine, un *double* e un *Double* è necessario, visto il significato che si vuol dare all'operatore, che il primo argomento sia ricevuto per riferimento in memoria. Occorre che si ripeta perché? Non dimenticate, appunto, che *"non fa scïenza, / sanza lo ritenere, avere inteso"*; invece la restituzione per riferimento non è obbligatoria, ma è altamente auspicabile, vista la forma della ricezione;
11. le funzioni che rappresentano la divisione prevedono una minima cautela contro l'eventualità che il secondo operando sia zero: non si tratta della migliore gestione possibile perché si attiva SOLTANTO in caso di identità esatta con la costante 0.0; ci si potrebbe mettere in guardia anche in presenza di divisori estremamente piccoli, anche se non nulli, o estremamente più piccoli rispetto al dividendo. Si osservi altresì che, in caso di tentata divisione per zero, la scelta del programmatore è stata di lasciare INTATTO il dividendo (per gli operatori con assegnamento) e di dare all'espressione il valore del dividendo (per *operator/*): in sostanza si attribuisce tacitamente al divisore NULLO il valore 1 (si tratta, ovviamente, di una scelta del tutto opinabile), ma contemporaneamente si ACCENDE la variabile membro booleana *errore*, che infatti viene proficuamente sfruttata in *main*;
12. la definizione dell'UNICO costruttore della classe consente, attraverso TUTTO il codice, la dichiarazione, con immediata inizializzazione tramite assegnamento, di oggetti della classe, come si potrebbe fare per un volgare intero. Questo fatto è GENERALE: se una classe dispone di un costruttore che, al netto di eventuali argomenti standard, ne abbia SOLO UNO effettivo, ALLORA nella dichiarazione di oggetti di quella classe è possibile usare la sintassi qui adottata per l'immediata inizializzazione, se a destra dell'operatore di assegnamento compare un oggetto del tipo atteso dal costruttore; e siccome ci possono essere quanti si vogliano costruttori in *overload*, questo significa che ci possono essere quante si vogliano immediate inizializzazioni DIVERSE per una sola classe (GRAN LINGUAGGIO);
13. se si volesse escludere un costruttore monoparametrico dalla facoltà di consentire immediate inizializzazioni del tipo proposto nell'esempio, all'atto della sua **dichiarazione** gli si premetterà la parola di vocabolario ***explicit***. Provate a farlo e contate gli errori che ne ricaverete (**FATELO!**);
14. il metodo pubblico *Errato( )* è un eminente esempio di funzione con proprietà quantistiche, ossia tale da influenzare lo stato di un'osservabile (la variabile *errore*) nel momento stesso in cui se ne effettua la misura. Infatti tale funzione non solo restituisce il valore della variabile in questione, ma la pone inesorabilmente nel suo stato iniziale (*false*), indipendentemente dal valore restituito;
15. nella funzione *Double exp(Double)* il valore 1.0 con cui si inizializza l'oggetto *r* che poi viene restituito è irrilevante; un'inizializzazione con un valore arbitrario è tuttavia necessaria perché la classe **non dispone di un costruttore di default**: provate a togliere *= 1.0* e vedrete (**FATELO!**);
16. invece l'identica inizializzazione per l'argomento ricevuto lo trasforma in un argomento *standard*, che rende possibile l'invocazione senza argomenti nell'ultima riga di *main* (il che, incidentalmente, garantisce che ad essere eseguita è proprio la *exp* che è stata scritta, perché quella dichiarata in *cmath* NON PUÒ ESSERE INVOCATA SENZA ARGOMENTI) e la veridicità dell'affermazione che ne precede il risultato sullo *standard output stream*;
17. la funzione *operator* con cui si chiude la definizione della classe consente il *casting* al tipo nativo *double* per un oggetto *Double*: il suo metodo di lavoro è lampante nella sua semplicità; se ne apprezzi piuttosto la sintassi, che è quella di un operatore unario, quale è, infatti, in cui nome e tipo restituito coincidono, un po' come accade con i costruttori. Il programma se ne giova, ad esempio, nella sottoespressione  
    *cout << (double)quota % Double(tasso);*  
    in cui la natura dei due operandi *quota* e *tasso* viene stravolta e trasformata in quella dell'altro operando, di modo che, ad essere eseguita, sia proprio la funzione contraria, ossia la *friend operator%(double, Double)*, a quella naturale, ossia il metodo membro *operator%(double)* ...GRAN LINGUAGGIO...specialmente considerando che si potrebbe introdurre un simile *operator* per qualsiasi tipo nativo... E anche NON nativo, cortocircuitando *cum grano salis* gli operatori di *casting* C++ e la rigidità delle loro applicazioni;
18. la prima riga di *main*, con una sintassi che si incontra per la prima volta durante il presente percorso, dichiara e immediatamente inizializza un cosiddetto puntatore a un membro di una classe, e specificamente un puntatore a un metodo membro senza argomenti e che restituisca un *double*: per questo può avere successo l'immediata inizializzazione che avviene, per l'appunto, attraverso il cosiddetto *offset* del metodo pubblico *rendi* all'interno della definizione della classe. Naturalmente BISOGNA esercitare la virtù della coerenza, e in effetti quel metodo restituisce proprio un *double* e NON RICHIEDE argomenti.  
    Il significato dell'operatore unario *&* nel presente contesto è appunto restituire un *offset* piuttosto che un indirizzo; del resto, come si è già detto, il nome di una funzione sarebbe, di suo, già un indirizzo e NON AVREBBE BISOGNO dell'operatore. Il fatto è che, quando viene compilata la riga in questione, NON ESISTE ALCUN INDIRIZZO DA PRENDERE per il metodo *rendi*, perché NON È STATO ANCORA REALIZZATO ALCUN OGGETTO *Double* e il metodo *rendi* non è qualificato *static* (ricordate? **RICORDATE!**).   
    Tuttavia esiste appunto uno scostamento ben definito del metodo *rendi* a partire dall'inizio della definizione della classe e dato essenzialmente dall'ordine topologico di inserimento dei membri nella definizione stessa. Allorché un oggetto (**OGNI** oggetto) della classe *Double* sarà istanziato, il DI LUI metodo membro *rendi* acquisirà AUTOMATICAMENTE l'indirizzo in memoria che si trova ESATTAMENTE allo scostamento sopra citato, a partire dalla collocazione in cui l'oggetto in questione INIZIA a occuparla.  
    E poiché, come appare evidente nella successiva espressione *cout*, il puntatore al metodo membro viene sempre utilizzato a partire da un oggetto della classe, quello che serve a TROVARE ESATTAMENTE il metodo GIUSTO che si vuol fare eseguire non è altro che l'indirizzo in cui si trova l'oggetto, dato con ogni evidenza dal fatto che l'operando di sinistra dell'operatore .\* **È** un tale oggetto, cui si aggiunga l'*offset*, informazione contenuta appunto nel puntatore da prima ancora che l'oggetto nascesse; **GRAN LINGUAGGIO**...se non ve ne siete accorti, accade la stessa cosa di quando si passa dallo spazio affine allo spazio euclideo....
19. prestate attenzione alle sintassi di dichiarazione, di inizializzazione e di utilizzazione del puntatore *d*: quantunque sia nella dichiarazione sia nell'inizializzazione compaia (OBBLIGATORIAMENTE) il risolutore di ambito *Double::*, l'ambito di visibilità del puntatore *d* è quello di *main*, NON ASSOLUTAMENTE quello della classe: infatti si è fatto apposta a nominarlo *d*, così da mostrare apertamente la veridicità dell'affermazione appena enunciata, visto che non si genera alcun conflitto con l'omonima variabile membro privata *Double::d*. Se immaginaste, per un momento, di togliere il risolutore dalla dichiarazione, stareste dichiarando un normalissimo puntatore a funzione nell'ambito di *main*, che in nessun modo potrebbe avere qualcosa da spartire con qualsivoglia classe, e TANTO MENO essere inizializzato con un offset...Non parliamo poi di togliere il risolutore dall'inizializzazione, che ci farebbe cadere nell'oscenità di utilizzare un identificatore *undeclared*.  
    Notate che, quando viene utilizzato *d*, la sintassi adottata è apparentemente IDENTICA alla dereferenza di un puntatore membro; l'avverbio apparentemente non è ozioso: infatti la sintassi **NON È IDENTICA** perchè, se tale fosse, dovrebbe essere possibile inserire un numero arbitraro di spazi tra il punto e l'asterisco, mentre **NON SE NE PUÒ INSERIRE NEMMENO UNO**. E comunque, a causa delle proprietà di precedenza e associatività degli operatori, l'eventuale dereferenza di un puntatore membro andrebbe compiuta premettendo l'operatore unario all'espressione che accede al membro. La realtà è che la coppia di caratteri *.\** costituisce UN UNICO OPERATORE, dedicato a svolgere il compito che QUI svolge, e NON DUE OPERATORI DIVERSI SCRITTI UNO ACCANTO ALL'ALTRO; in altre parole UN PUNTATORE **A** MEMBRO è del tutto diverso da UN PUNTATORE MEMBRO: questo sì che è dichiarato nell'ambito della classe.
20. se tornate a leggere, qualche canto indietro, l'elenco di TUTTI gli operatori del linguaggio, troverete che QUESTO operatore era stato nominato, assieme al suo compare *->\**, da usare coerentemente quando alla sua sinistra si trova un puntatore a un oggetto piuttosto che un oggetto, MA la loro spiegazione era stata procrastinata. Ora quel futuro è transitato nel passato. Naturalmente è possibile dichiarare anche puntatori a membro che puntino semplici variabili piuttosto che metodi: la sintassi è la stessa, anzi è assai più semplice dato che non richiede uso di parentesi tonde in alcuna posizione. Altrettanto ovviamente non è possibile dichiarare e utilizzare puntatori per accedere a membri non pubblici: il linguaggio non ammette il motto, tipicamente e malauguratamente italiano, *"fatta la legge, trovato l'inganno"*...
21. nella funzione *main* compaiono numerosi oggetti senza nome: ce n'è uno diverso, e di valore sempre uguale, per OGNI INVOCAZIONE ESPLICITA del costruttore della classe, ossia per ogni occorrenza dell'espressione *Double(tasso)*; in un programma didattico come questo si può anche compiere qualche abuso, ma per quando sarete più evoluti sappiate fin da adesso che una simile pratica appesantisce il programma, perché questi oggetti innominati, pur non essendo mai più accessibili al di fuori dell'espressione in cui compaiono, permangono tuttavia nella memoria (e quindi la consumano) per tutto l'ambito della funzione in cui sono realizzati, finché non se ne esce (e quindi, trattandosi di *main*, per tutta la durata del programma). Va osservato che, nella sottoespressione  
     *cout << (tasso = (double)(Double(tasso) += quota));*   
      
    non è la variabile *tasso* a subire l'incremento di *quota* da parte dell'operatore *+=*, ma, per l'appunto, SOLO l'oggetto innominato *Double(tasso)* che esegue a volo la SUA funzione membro *operator+=(Double)*. La variabile *tasso*, che è un semplice *double*, si ritrova poi aggiornato coerentemente il PROPRIO valore SOLO grazie all'operatore di *casting* applicato all'oggetto senza nome restituito da *operator+=* e al successivo assegnamento: il ruolo delle parentesi è CRUCIALE in questa sottoespressione...;
22. ci sono altri due oggetti senza nome che invocano l'esecuzione del metodo *Errato( )*: uno è quello restituito da *operator/* e l'altro è quello restituito da *operator/=*. Entrambe queste funzioni sono eseguite tramite l'oggetto *quota* (cui appartengono), ma solo la seconda ha effetti collaterali sull'oggetto, come si riesce ad apprezzare quando, subito dopo, viene scritto sullo *standard output stream*;
23. serbate nella vostra mente (non è la prima volta che ne siete avvertiti) questa storia degli oggetti senza nome che possono comparire nell'ambito di una funzione...

# Canto ventisettesimo: anteprima, a volo di gabbiano, sulle classi *template*

*"Bella, l'idea della classe Double con cui poter usare anche operatori vietati al tipo nativo double; le voglio aggiungere anche le definizioni per gli operatori di bit masking (****&   |   ^****) e di scorrimento bit (****>>   <<****). Poi, quando l'avrò finita di scrivere, così come mi garba, ne farò un gigantesco copia-incolla e, nella copia, farò sostituire al text editor tutte le occorrenze di Double con Float e tutte quelle di double con float...In questo modo avrò, senza alcuna fatica aggiuntiva, anche una classe del tutto analoga, valevole per il tipo nativo float"*   
  
Così potrebbe argomentare, non senza buone ragioni, la mente dei più attivi tra voi; il pensiero così formulato è indubitabilmente fondato e vero, ma se poi ne volessimo un'altra ancora, valida per i *long double*, che si fa: un terzo copia-incolla? *"Perché no?"*, potreste obiettare...  
Perché si fa prima, e meglio, ad apportare al programma del canto precedente le seguenti varianti:

#include <iostream>  
#include <cmath>  
using namespace std;  
template <class XXX>  
class Double  
{bool errore;  
XXX d;  
public:  
  
Double<XXX> operator+(XXX s)  
{Double<XXX> r = d; r.d += s; return r;}  
  
Double<XXX> operator/(XXX s)  
{Double<XXX> r = d;  
r.errore = s == 0.0;  
if(!r.errore) r.d/=s;  
return r;}  
  
Double<XXX> operator-( )  
{d \*= -1.0; return \*this;}  
Double<XXX> operator+( )  
{return \*this;}  
  
Double<XXX> operator+(Double<XXX> d)  
{Double<XXX> r = this->d; r.d += d.d; return r;}  
  
Double<XXX> operator/(Double<XXX> d)  
{Double<XXX> r = this->d;  
r.errore = d.d == 0.0;  
if(!r.errore) r.d /= d.d;  
return r;}  
  
template <typename YYY> friend Double<YYY> operator+(YYY, Double<YYY>);  
  
Double<XXX> operator+=(XXX s)  
{d+=s; return \*this;}  
  
Double<XXX> operator/=(XXX s)  
{errore = s == 0.0;  
if(!errore) d/=s;  
return \*this;}  
  
Double<XXX> operator+=(Double<XXX> s)  
{d += s.d; return \*this;}  
  
Double<XXX> operator/=(Double<XXX> s)  
{errore = s.d == 0.0;  
if(!errore) d/=s.d;  
return \*this;}  
  
template <class YYY> friend YYY& operator+=(YYY&, Double<YYY>);  
  
XXX operator%(XXX d)  
{return d \* this->d / 100.0;}  
  
XXX operator%(Double<XXX> d)  
{return d.d \* this->d / 100.0;}  
  
template <class YYY> friend YYY operator%(YYY, Double<YYY>);  
  
template <class YYY> friend ostream& operator<<(ostream &, Double<YYY>);   
  
template <class YYY> friend YYY exp(YYY);  
  
Double(XXX s) : d(s) {errore = false;}  
  
bool Errato( )  
{if(errore) return !(errore=false);  
return errore;}  
  
XXX rendi( )  
{return d;}  
  
operator XXX( )  
{return d;}  
}; // fine della definizione della classe (completata)  
  
// definizione delle funzioni friend (FUORI dalla classe!)  
  
template <typename XXX>  
Double<XXX> operator+(XXX d, Double<XXX> D)  
{Double<XXX> r = d; r.d += D.d; return r;}  
template <typename XXX>  
XXX& operator+=(XXX &d, Double<XXX> D)  
{d += D.d; return d;}  
template <typename XXX>  
XXX operator % (XXX d, Double<XXX> D)  
{return D.d \* d / 100.0;}  
template <typename XXX>  
ostream& operator<<(ostream &o, Double<XXX> D)  
{return o << D.d;}  
template <typename XXX>  
XXX exp(XXX x)  
{XXX r = 1.0; r.d = exp(x.d); return r;}  
  
int main( )  
{  
using T = double;  
using D = Double<T>;  
T tasso = 12.8, zero = 0, due = 2, uno = 1;  
D quota = 34567.789;  
T (D::\*d)(  ) = &D::rendi;  
cout << "posso usare il puntatore a funzione membro \"d\""  
"\nper eseguire la funzione rendi e ottenere il valore della"  
"\nvariabile privata di un oggetto Double che risulta "  
<< (quota .\* d)(  ) << '\n';  
cout  
<< "posso eseguire double + Double: "  
<< tasso + quota << '\n';  
cout  
<< "ma anche Double + double: "  
<< quota + tasso << '\n';  
cout  
<< "e financo Double + Double: "  
<< quota + D(tasso) << '\n';  
cout  
<< "poi posso calcolare il "  
<< tasso << "% di " << quota  
<< "\nsia come Double % double: "  
<< quota % tasso << '\n';  
cout  
<< "sia come double % Double: "  
<< (T)quota % D(tasso) << '\n';  
cout  
<< "sia come Double % Double: "  
<< quota % D(tasso) << '\n';  
cout  
<< "Posso constatare se funziona la divisione\n";  
if((quota / zero) . Errato( ))  
cerr << "PIRLA! non si divide per 0\n";  
if(!(quota /= due) . Errato( ))  
cout  
<< "per 2 invece puoi: "  
<< quota << '\n';  
cout  
<< "Posso anche incrementare "  
<< quota << " di " << tasso;  
cout  
<< " ottenendo " << (quota += tasso) << '\n';  
cout  
<< "e anche incrementare " << tasso  
<< " del valore incrementato di " << quota  
<< " ottenendo "; cout << (tasso += quota) << '\n';  
cout  
<< "posso applicare il - unario a "  
<< quota << " ottenendo ";  
cout << -quota << '\n';  
cout  
<< "e incrementare con questo il valore attuale di "  
<< tasso   
<< "\nlavorando con operator+= tra due Double, e ottenendo ";  
cout  
<< (tasso = (T)(D(tasso) += quota)) << '\n';  
cout  
<< "infine posso calcolare l'esponenziale di un Double come "  
<< D(tasso) << " ottenendo "  
<< exp(D(tasso)) << '\n'  
<< "e anche scrivere il valore del numero di Nepero che è "  
<< exp(D(uno)) << '\n';  
}

Se eseguirete QUESTO programma (**ESEGUITELO, DUNQUE!**) vedrete che produrrà uno *standard output stream* **IDENTICO** a quello del programma proposto nel canto 26-esimo; ma QUESTO, senza più modificare **NULL'ALTRO** che la linea di *main* che recita  
*using T = double;*,  
trasformandola, ad esempio, in *using T = float;* oppure in *using T = long double;* e PERFINO in *using T = int;* (con tutti i modificatori di *int* **plausibili**, e con la sola eccezione della funzione *exp*, che andrebbe curata in qualche modo, dato che NON ha un *overload* per argomento intero) può essere rieseguito con successo per OGNUNA di tali modestissime modifiche (GRANDISSIMO LINGUAGGIO).  
  
Un tale stupefacente risultato è stato ottenuto attraverso la trasformazione della classe *Double* in una cosiddetta *template class*. Nel prossimo canto il gabbiano che, nel titolo di questo, sta volando alto, contemplando da lunge *il gran padre Oceàno* delle *template class*, si abbasserà di quota e giungerà ad ammarare in mezzo a loro, per raccontare a voi quanti buoni pesci riuscirà a pescarvi.  
Per adesso accontentatevi di annotare l'uso della parola *using* (il bisticcio di parole mi garbava) adottato appunto da *main* per creare degli *alias* di tipi già definiti da poter usare tranquillamente in ogni contesto occorressero.

**Canto ventottesimo: il gabbiano a livello del mare *template***

Il linguaggio C++ contempla il fenomeno della templatizzazione (mi sia concesso quest'orribile neologismo) sia per le classi sia per le funzioni: nell'un caso e nell'altro si premette COMUNQUE alla normale sintassi per la dichiarazione di una classe o di una funzione una clausola che ha il seguente formato

template <*lista di parametri templatizzati*>

ove *template* è una parola del vocabolario del linguaggio, i segni *<* e *>* sono obbligatori ed è invalso l'uso di denominarli, in questo contesto, parentesi angolari, per sottolineare la necessità del loro bilanciamento, e con *lista di parametri templatizzati* s'intende un elenco costituito da ZERO o più elementi separati con virgole, ovviamente quando sono almeno DUE.  
Ogni elemento della *lista* può avere uno dei seguenti aspetti:

1. un identificatore, confinato all'ambito della classe o funzione, preceduto dalla parola di vocabolario *class*
2. un identificatore, confinato all'ambito della classe o funzione, preceduto dalla parola di vocabolario *typename*
3. una normale dichiarazione di una variabile, confinata all'ambito della classe o funzione, con alcune restrizioni sul tipo cui può essere fatta appartenere: si parla, in questo caso, di un parametro templatizzato NON tipo;
4. un'altra clausola *template*, fino a qualsiasi (ragionevole) livello di nidificazione: si parla, in questo caso, di *template-template parameter*;
5. SOLO in ULTIMA, o eventualmente UNICA, posizione un cosiddetto *parameter pack*, o più di uno solo; vale a dire un identificatore, confinato all'ambito della classe o funzione, preceduto da tre puntini consecutivi a loro volta preceduti da *typename* o da *class* (cfr. punti 1. e 2.) ovvero da un nome di tipo tra quelli consentiti in una clausola di templatizzazione NON tipo.

**Osservazione 1:**  
le forme 1. e 2. del precedente elenco sono fra loro del tutto interscambiabili: la differenza fra *typename* e *class* si riscontra in altri contesti.   
  
**Osservazione 2:**  
È possibile assegnare valori *standard* agli elementi della *lista*, con la stessa sintassi adottata per gli argomenti *standard* delle funzioni, ma, come colà, se un elemento della *lista* ha un valore *standard*, allora lo devono avere ANCHE TUTTI GLI EVENTUALI ELEMENTI SUCCESSIVI, con l'eccezione, peraltro ovvia, di un eventuale *parameter pack* conclusivo.   
  
**Osservazione 3:**  
Quando la *lista* è vuota (rileggere: contiene ZERO o più elementi) si parla di specializzazione di una classe *template* già dichiarata in precedenza.   
  
**Osservazione 4:**  
la specializzazione di una classe *template* può anche essere parziale, nel senso che si specificherà appresso.   
  
**Osservazione 5:**  
Tutti gli identificatori di cui si parla nell'elenco possono essere omessi se la classe/funzione è SOLO dichiarata, ma non definita, esattamente come avviene per i nomi degli argomenti nella dichiarazione, senza definizione, di una funzione ordinaria.   
  
**Osservazione 6:**  
Quando, nell'ultimo punto dell'elenco, si scrive più di un solo, occorre che sia ben chiaro che cosa si intende. Tanto per cominciare più di un solo *parameter pack* si può riscontrare soltanto nelle funzioni *template*, che ne permettono la deduzione, e NON nelle classi *template*; inoltre, anche per le funzioni *template*, bisogna aver cura di evitare ogni ambiguità nell'espansione dei *pack*, pertanto:

* i "più di un solo *pack*" potranno essere, al massimo, DUE;
* o si tratta di un *pack* di NON TIPI seguito da un *pack* di TIPI;
* oppure si tratta di due *packs*, entrambi di TIPI, distinguibili uno dall'altro tramite qualche forma di "frontiera" NON AMBIGUA che li separi nettamente.

**Esempi:**

1. la più semplice delle definizioni di una classe *template*, come quella del canto precedente:  
   *template <class X> class Ciccio {/\*omissis\*/};*
2. la precedente, seguita da una sua specializzazione  
   *template <class X> class Ciccio {/\*omissis\*/};  
   template < > class Ciccio<int> {/\*omissis\*/};*
3. una definizione con lista contenente più di un solo elemento:  
   *template <class X, class Y, int z> class Ciccio {/\*omissis\*/};*
4. la precedente, seguita da due sue specializzazioni parziali:  
   *template <class X, class Y, int z> class Ciccio {/\*omissis\*/};  
   template <class X> class Ciccio<X, int, z> {/\*omissis\*/};  
   template <class X, class Y> class Ciccio<X, Y, 148> {/\*omissis\*/};*
5. la precedente, con alcuni valori *standard*:  
   *template <class X, class Y=int, int z=12> class Ciccio {/\*omissis\*/};*
6. una definizione che fa uso di un *parameter pack*:  
   *template <typename ... P> class Ciccio {/\*omissis\*/};*
7. una definizione che fa uso di un *template-template parameter*, con valore *standard*, ovviamente OPZIONALE:  
   *template <class X> class Pappo {/\*omissis\*/};  
   template <class X> class Peppo {/\*omissis\*/};  
   template <template <typename> class Y=Pappo> class Ciccio {/\*omissis\*/};*
8. e una che fa uso di un *template-template parameter* due volte nidificato, ancora con valore *standard*:  
   *template <template <typename> class X> class Puppo {/\*omissis\*/};  
   template <template <template<typename> class> class Z=Puppo> class Ciccio {/\*omissis\*/};*
9. miscellanea:  
   *template <int N, class A, class B = double, typename ... P>  
   class Ciccio {/\*omissis\*/};*
10. una funzione *template*, piuttosto che una classe:  
    *template <int N, class A, class B = double, typename ... P>  
    A    funza (A a, B b, P ... p) {A r; /\*omissis\*/ return r;}*

Dopo una così vasta panoramica, si spera sufficiente, sulle sintassi da adottare per la definizione/dichiarazione di classi e funzioni *template*, resta da capire di che cosa si stia parlando; né le funzioni né le classi *template* sono vere funzioni o vere classi: si tratta solo di modelli (*template*, appunto...) delle une e delle altre.  
  
È come quando, in un *atelier* di *haute couture* (questa similitudine dovrebbe essere ben compresa specialmente dalle ragazze, fino a prova contraria), trovaste i modelli cartacei degli abiti dell'ultima collezione: nessuno penserebbe di indossarli così come sono, per quanto follìe del genere possano anche essere accadute, nel mondo reale..., fino a quando non arriva un sarto con un paio di forbici e una pezza di tessuto, che ritaglierà seguendo il modello, per confezionare un abito vero che realizzi QUEL MODELLO nel tessuto che egli ha scelto.  
Altri sarti potranno usare tessuti differenti, e, compatibilmente con quello che ne capisco io di alta moda, ossia NIENTE, lo stesso modello potrebbe servire a confezionare tanto un cappotto invernale quanto uno spolverino estivo...  
  
In modo analogo, una *template* del linguaggio C++, classe o funzione che sia, è solo il modello cartaceo di TUTTO quello che potrà essere una volta che qualcuno (il sarto, ossia il programmatore) lo realizzi nella concretezza di un tessuto, ossia della specificazione esplicita dei tipi effettivi che dovranno essere usati per la creazione di abiti... volevo dire... di OGGETTI, se si parla di classi.  
  
È quanto accaduto con la classe *template* del canto precedente e con tutte le funzioni *template* ad essa associate come *friend*: solo nel momento in cui il programmatore, nella funzione *main*, richiede che siano realizzati oggetti conformi al modello, dichiarando di quale tessuto debbano essere fatti, la classe prende la sostanza di quel tessuto e l'oggetto viene istanziato; e per essere ancor più specifici, il *main* prende tale iniziativa nella riga

D quota = 34567.789;

la quale, stante il significato di *D*, desunto appena due righe sopra, viene letta dal compilatore come se fosse stata scritta

Double<double> quota = 34567.789;

e insegna a chiunque come si istanzia un oggetto secondo una classe *template* e come si specifica il tessuto.  
  
In parole ancora più semplici e dirette si può concepire nella mente una *template* come una classe o una funzione in cui PERFINO IL TIPO DELLE VARIABILI MEMBRO O DEGLI ARGOMENTI O DEL VALORE RESTITUITO è una VARIABILE il cui nome è quello specificato nella *lista*, dopo *typename* o *class*, e il cui valore viene assegnato dal programmatore O attraverso i valori *standard* provvisti nella definizione della *template*, O, in assenza o sostituzione di questi, all'atto dell'istanziazione di un oggetto, se si tratta di una classe, o della richiesta di esecuzione, se si tratta di una funzione.  
  
Per giunta, quando nella *lista* della *template* compare un *parameter pack* ADDIRITTURA È UNA VARIABILE, implicitamente di tipo *int*, IL NUMERO DI TIPI **VARIABILI** PREVISTI PER LE VARIABILI MEMBRO O PER GLI ARGOMENTI e il valore di questa variabile implicita, ossia quanti TIPI VARIABILI sussistono, è desunto dalla cosiddetta *pack expansion*, che si discuterà ben presto.  
  
**FORMIDABILE LINGUAGGIO!** Riuscite a intravedere la possibilità di compiere QUALSIASI AZIONE, e quando dico QUALSIASI intendo **QUALUNQUE**, usando **UNA, SOLA, UNICA** interfaccia?  
  
Altrimenti detto: riuscite a capire quanti crostacei, molluschi, celenterati, pesci e mammiferi marini potrà prendere il nostro gabbiano ammarato, dalla *Euphasia superba* (krill) alla *Balaenoptera musculus* (balenottera azzurra), usando sempre lo stesso, identico, UNICO ... suo becco?  
  
Non ci riuscite?  
  
Vi fa ampiamente difetto l'ottava virtù del buon programmatore...  
  
Si tenterà di farvelo capire almeno un po' nel prossimo canto; prima di chiudere questo, però, mette conto sottolineare dettagliatamente il comportamento del compilatore allorché si trova di fronte alla definizione di una *template*: si limita ESCLUSIVAMENTE al controllo sintattico del codice, **SENZA GENERARE ALCUN CODICE BINARIO CHE NE SIA LA TRADUZIONE**.  
E del resto, come potrebbe? Un modello cartaceo non pesa sulla bilancia...quale sarebbe il suo *sizeof*? Un conto è realizzarlo in organza, un altro in juta ...  
Il codice binario viene pertanto prodotto SOLO SE e SOLO QUANDO viene fornito il materiale di costruzione, ossia, ancora come sopra, nel momento dell'istanziazione di un oggetto, o della richiesta di esecuzione, per cui il codice binario NON SIA STATO ANCORA GENERATO nella sessione di compilazione corrente.  
  
Ciò significa, in altri termini, che una *template* può essere tradotta in binario, durante la compilazione di un intero programma, da un minimo di ZERO volte, se non venisse usata mai, a un massimo di volte che è pari al numero di istanziazioni DIVERSE che di quella *template* si richiedono.  
  
Attenzione, però, a non illudersi di poterne combinare più di Carlo in Francia, nella scrittura di una *template*, col pretesto che *"tanto non si usa"*...il controllo della sintassi c'è comunque.  
  
A proposito di quest'ultima affermazione, mette conto effettuare qualche minuscolo approfondimento riguardante il problema dei cosiddetti *nomi dipendenti*: in anglo-americano *"dependent names"*. Qualche cenno riassuntivo su di essi si trova nella pagina dedicata alla parola di vocabolario *template*, ma qui ne verrà ora data una trattazione più completa.  
  
Per prima cosa bisogna capire di che cosa si sta parlando e per quale ragione potrebbero provocare qualche problema: un nome dipendente è tale in quanto dipende da uno o da qualche parametro della clausola di templatizzazione; ad esempio nell'ambito della seguente funzione  
  
*template <class X> void funza(  )  
{X::x \* x; /\* omissis\*/}*   
  
il nome *X::x* è appunto un nome dipendente dal parametro di templatizzazione *X* e assumerà significati diversissimi secondo quale *funza* venga istanziata.  
Se nella stessa sorgente di codice si trovassero definite le due seguenti classi  
  
*struct A {typedef int x;};  
struct B {static const int x = 1;};*   
  
e fosse anche dichiarata una variabile globale *int x(1)*, prima della definizione di *funza*, quale significato avrebbero le due seguenti istanziazioni di *funza*?  
  
*funza<A>(  );  
funza<B>(  );*   
  
Credo che si possa ora capire dove sta il problema: nella prima istanziazione il nome dipendente nell'ambito di *funza* si risolve in una dichiarazione di un puntatore a *int* chiamato *x*, mentre nella seconda diverrebbe un'espressione moltiplicativa senza effetti collaterali tra due variabili intere: la *x* dell'ambito globale e l'omonimo membro *static* della classe *B*.   
  
È del tutto evidente che, secondo quanto fosse scritto negli *omissis*, si potrebbe produrre un numero di errori superiore a qualsiasi biasimo.  
  
Per questa ragione lo *standard* del linguaggio pretende chiarezza da parte del buon programmatore, esortandolo ad assumersi personalmente la responsabilità di indicare al compilatore se il *nome dipendente* è o NON è quello di un TIPO; il compilatore, da parte sua, congettura che **NON LO SIA**, a meno che, appunto, il programmatore non lo affermi esplicitamente, premettendo al *nome dipendente* la parola di vocabolario *typename*, e scriva dunque *funza* in questo modo alternativo:  
  
*template <class X> void funza(  )  
{typename X::x \* x; /\* omissis\*/}*   
  
Per quel che è stato detto in questo stesso canto va ancora una volta sottolineato che il compilatore segnala errore **solo all'atto della richiesta di un'istanziazione che produca esiti contraddittorii**: in assenza dell'istanziazione *funza<A>(  );* la versione priva della parola *typename* viene compilata allegramente, alla peggio con la segnalazione che si è scritta un'espressione inutile.   
  
Ci può essere una situazione anche più complessa, come quella contemplata dal seguente codice, e da ogni altro codice che reiteri e nidifichi la stessa situazione su livelli di profondità superiore:  
  
*# include <iostream>  
using namespace std;  
  
template <class X> struct Pappo {  
template <class Y> struct Peppo{ // una classe template nidificata  
typedef double questo\_deve\_essere\_un\_tipo;  
};  
struct questo\_deve\_essere\_un\_tipo // una classe ordinaria nidificata  
{template <class W> void gunza(  )  
{cout << "cucu\n";}}; };  
  
template <class Z> void funza(  ) // una funzione template  
{  
typename Pappo<Z>::questo\_deve\_essere\_un\_tipo var;  
typename Pappo<Z>::template Peppo<int>::questo\_deve\_essere\_un\_tipo d;  
var.template gunza<int>(  );  
}  
  
int main(  ) {  
funza<double>(  ); }*   
  
Se si osserva con attenzione il contenuto dell'ambito di *funza* vi si trovano i seguenti *nomi dipendenti*:

* *Pappo<Z>::questo\_deve\_essere\_un\_tipo*   
  che dipende direttamente dal parametro di templatizzazione *Z* e, come si evince alla semplice lettura, "deve essere un tipo": per questo è prefissato con *typename* in quella che deve poter essere la dichiarazione dell'oggetto *var*;
* *Peppo*   
  che dipende da *Z* indirettamente, in quanto membro di *Pappo<Z>* e ivi dichiarato come nome di classe *template*: per questo tale parola di vocabolario viene premessa all'istanza *Peppo<int>*. In sua assenza si sarebbe avuta un'ambiguità d'interpretazione del successivo carattere <;
* *Peppo<int>::questo\_deve\_essere\_un\_tipo*   
  che dipende indirettamente da *Z* attraverso l'istanza *Peppo<int>* di *Peppo*, e, come già il suo omonimo membro di *Pappo<Z>* deve essere prefissato con *typename* per poter portare a compimento la corretta dichiarazione di *d*;
* *gunza*   
  che dipende indirettamente da *Z* in quanto membro di *var*, istanza di *Pappo<Z>::questo\_deve\_essere\_un\_tipo*, essendo ivi definita come una funzione *template*: per questo tale parola la precede, alla stessa stregua di quanto era accaduto per *Peppo<int>*.

Ora prestate attenzione a quest'altra situazione:  
  
*# include <iostream>  
using namespace std;  
  
struct Pappo {  
void gunza(  )  
{cout << "cucu\n";}  
template <class Y> void gunza(  )  
{Y y(22);  
cout << "coco " << y << '\n';}  
};  
  
template <class X> struct Peppo {  
void gunza(  )  
{cout << "dudu\n";}  
template <class Y> void gunza(  )  
{Y y(33);  
cout << "dodo " << y << '\n';}  
};  
  
template <class Z> void funza(  )  
{  
Z var;  
Peppo<Z> vor;  
var.gunza(  );  
var.template gunza<int>(  );  
vor.gunza(  );  
vor.template gunza<int>(  );  
}  
  
int main(  ) {  
funza<Pappo>(  );  
funza<Peppo<Pappo>>(  );  
}*   
  
Nessuno potrà negare che tanto nell'ambito di *funza* quanto in quelli delle due *template* omonime membri di *Pappo* e *Peppo* vi siano dei nomi dipendenti, nel senso che si è definito: precisamente il nome *Y* in entrambe le *gunza* e i nomi *Z* e *Peppo<Z>* in *funza*.  
  
Tuttavia NESSUNO di questi è stato prefissato con *typename*, ANZI se provaste a metterli incorrereste in un errore per ognuno che ne aggiungeste.  
  
La ragione sta nel fatto che questi nomi sono riconosciuti come nomi di tipi dal compilatore, e senza la minima ombra di dubbio, già all'atto dell'analisi sintattica del codice, nella cosiddetta **istanziazione corrente**.   
  
Ciò significa che quando il compilatore affronta questa sorgente di programma si rende immediatamente conto, ad esempio compilando le due *gunza*, che il nome *Y*, in qualunque istanziazione di *gunza* possa aver luogo, sarà sempre e comunque quello di un tipo, dato che coincide col parametro di templatizzazione, altro che dipenderne; e anche per quanto riguarda *Peppo<Z>* il compilatore non può nutrire alcun dubbio sulla natura di tale nome, perché ha appena visto che *Peppo* è una classe *template* con UN tipo templatizzato e che *Z* è per forza, appunto, un tipo come era *Y* nelle *gunza*.  
  
In definitiva i dubbi gli possono venire quando c'è di mezzo un risolutore di ambito, come avveniva nel codice precedente...

**Canto ventinovesimo: a caccia e a pesca nel mar dei *template***

Il programma del canto 27, come si è detto, fornisce un'immagine molto pallida di quanto si possa ottenere grazie alla *templatizzazione*, e tuttavia già sufficiente a lasciarne intuire l'utilità e a descriverne le sintassi meno astruse.  
  
Senza pretendere di esaurire l'argomento, la qual cosa, restando sulla similitudine marina, richiederebbe di calarsi fino a profondità superiori a quelle dell'abisso Challenger, si esaminerà in questo canto un programma che, almeno, fa scendere fino al limite consentito dalle bombole.  
  
Eccolo qua:

# include <iostream>  
# include <tuple>  
# include <typeinfo>  
# include <boost/variant.hpp>  
using namespace std;  
using namespace boost;  
  
template <size\_t n, typename ... T>  
variant<T...> vget(size\_t i, const tuple<T...>& t)  
{  
if(i == n) return get<n>(t);  
else if(n == sizeof...(T) - 1) throw "Tupla esaurita.\n\n";  
else return vget <(n < sizeof...(T)-1 ? n+1 : 0)> (i, t);  
}  
  
template <typename ... T>  
variant<T...> dget(size\_t i, const tuple<T...>& t)  
{  
return vget<0>(i, t);  
}  
  
template <class X> struct Pappo  
{X x;  
Pappo(  ) : x(12) {clog << "Pappo(  ) ";}  
Pappo(int i) : x(i) {clog << "Pappo(int) ";}  
Pappo(double i) : x(i) {clog << "Pappo(double) ";}  
};  
  
template <class X> struct Peppo  
{X x;  
Peppo(  ) : x(24) {clog << "Peppo(  ) ";}  
Peppo(int i) : x(i) {clog << "Peppo(int) ";}  
Peppo(double i) : x(i) {clog << "Peppo(double) ";}  
};  
  
template <template <typename> class X>  
struct Puppo  
{  
X<double> x;  
Puppo(  ) : x(36) {clog << "Puppo(  ) ";}  
Puppo(int i) : x(i) {clog << "Puppo(int) ";}  
Puppo(double i) : x(i) {clog << "Puppo(double) ";}  
Puppo operator+(int l) {return \*this;}  
};  
  
template <  > struct Puppo<Peppo>  
{  
Peppo<int> x;  
Puppo(  ) : x(36) {clog << "Puppo<Peppo>(  ) ";}  
Puppo(int i) : x(i) {clog << "Puppo<Peppo>(int) ";}  
Puppo(double i) : x(i)   
{clog << "Puppo<Peppo>(double) ";}  
Puppo operator+(int l) {return \*this;}  
};  
  
template  
<typename A, class B = double, int N = 5,  
template <typename> class Y = Pappo,  
template <template<typename> class>  
class Z = Puppo,  
class ... Altri>  
struct Ciccio  
{  
A a;  
B b;  
Y<A> y;  
Z<Pappo> z;  
int k;  
void info(  )  
{  
clog << "è stato istanziato un oggetto di tipo\n"  
<< typeid(\*this).name(  )  
<< "\ntale oggetto contiene:\n"  
<< "una variabile di tipo " << typeid(k).name(  )   
<< " e di valore " << k << '\n'  
<< "una variabile di tipo " << typeid(a).name(  )  
<< " e di valore " << a << '\n'  
<< "una variabile di tipo " << typeid(b).name(  )   
<< " e di valore " << b << '\n'  
<< "una variabile di tipo " << typeid(y).name(  )   
<< " e di valore " << y << '\n'  
<< "una variabile di tipo " << typeid(z).name(  )  
<< " e di valore " << z << '\n'  
<< "e una funzione membro variadica che agisce in questo modo:\n";  
}  
void f(Altri ... quq)  
{  
tuple<Altri...> Tupla{quq...};  
int n = 0;  
auto Tupla\_con\_almeno\_un\_elemento =  
tuple\_cat(tie(n), Tupla);  
if(!sizeof...(Altri))  
cout << "non ho ricevuto ALCUN ARGOMENTO\n";  
for(size\_t i=1; ; ++i)  
try{  
auto t = dget(i, Tupla\_con\_almeno\_un\_elemento);  
cout << "ho ricevuto il parametro " << t << '\n';}  
catch(const char \* c)  
{cout << c << endl; break;}  
}  
Ciccio(  ) : k(N), a(1), b(2), y(27), z(68)  
{clog << "Ciccio(  ) ";}  
};  
  
template < template<typename> class A >  
ostream& operator<<(ostream & o, A<int> a)  
{  
o << "(scritto da operator << # 1): ";  
return o << a.x;  
}  
  
template < template<typename> class A >  
ostream& operator<<(ostream & o, A<double> a)  
{  
o << "(scritto da operator << # 2): ";  
return o << a.x;  
}  
  
template   
< template < template <typename> class > class A >  
ostream& operator<<(ostream & o, A<Pappo> a)  
{  
o << "(scritto da operator << # 3): ";  
return o << a.x;  
}  
  
template  
<typename A, class B = double, int N = 5,  
template <typename> class Y = Pappo,  
template <template<typename> class>  
class Z = Puppo,  
class ... Altri>  
auto funza(A a, B b, Altri ... p) -> decltype(a)  
{A r(a); return r;}  
  
int main(  )  
{  
Ciccio<int, int, 4, Peppo, Puppo, double, int, char\*> ciccio;  
ciccio . info(  );  
ciccio.f(1.4, 1, const\_cast<char \*>("cucu"));  
Ciccio<double> coccio;  
coccio . info(  );  
coccio.f(  );  
cout   
<< "nel programma è definita anche una funzione template variadica"  
"\nche può essere eseguita in una quantità INDUSTRIALE di modi diversi\n"  
"e può svolgere, di conseguenza, CHISSÀ CHE OPERAZIONI...come si arguisce sotto\n";  
cout << "modo 1 " << funza(11, 22) << '\n';  
cout << "modo 2 " << funza(14.5, 'c') << '\n';  
Peppo<int> peppo;  
Puppo<Pappo> puppo;  
cout  
<< "modo N "  
<< funza("coco", "cici", 333, peppo, puppo, 11, 31, "tanto va la gatta al lardo\n");  
cout << "modo N+1" << funza(puppo, peppo, "zeta") << '\n';  
}

Prima di demoralizzarsi e/o annusare sali per rinvenire **ESEGUITE IL PROGRAMMA SENZA INDUGIO**: se vi manca il pacchetto *boost*, accessorio per il compilatore, lo si installi, perché male non vi fa di certo. Qualora non ci riusciste, ma ne devo dubitare, sollevate il problema in aula. In ogni caso, di tale pacchetto si potrebbe fare a meno, al solo prezzo di cambiare la definizione della funzione *dget*.  
  
**Discussione e spiegazione del programma:**

1. la coppia di funzioni *vget* e *dget*, con cui il programma esordisce, sono la sola dipendenza del codice dal pacchetto *boost*: nel documento incluso *boost/variant.hpp* si trova appunto definita la classe *template* variadica, ossia contenente nella sua *lista template* un *parameter pack*, denominata *variant*, della quale entrambe le funzioni restituiscono l'istanza di un oggetto. Entrambe ricevono anche, per riferimento, un'istanza della classe *template* variadica *tuple*, definita nel documento incluso omonimo.
2. se si legge attentamente il codice delle due funzioni citate, si vedrà facilmente che *vget* è potenzialmente ricorsiva e che viene innescata da *dget*. Per comprendere il significato di tutto l'*ambaradan* è necessario prima capire che cosa siano gli oggetti *tuple* e *variant*: *tuple* è descritta altrove in queste pagine, specificamente nella descrizione della parola *template*, ma ADESSO NON LA CERCATE; essenzialmente si tratta di un contenitore di elementi eterogenei nel tipo, in qualche modo simile, ma NON UGUALE, a una classica *struct* priva di metodi e di membri statici; in modo analogo un oggetto di tipo *variant* può essere assimilabile a una classica *union*: ciò che *tuple* e *variant* hanno di profondamente DIVERSO rispetto alle *struct* e *union* classiche è appunto la loro variadicità.
3. La funzione *dget*, che innesca *vget*, è invocata dal metodo *void f(Altri ... p)* appartenente alla classe *template* variadica *Ciccio*. Si tratta in effetti di una *template struct*: la scelta è stata fatta in modo da non doversi preoccupare anche dell'eventuale esistenza di membri non pubblici, ma tutto avverrebbe in modo equivalente anche per una *class*. Il metodo *f* è, a propria volta, un membro variadico, come si desume dall'argomento che riceve, vale a dire un *parameter pack*, opportunamente inserito all'ultimo posto della *lista di template* della classe di appartenenza. Quando invoca *dget*, le trasmette una *tuple* costruita localmente nell'ambito di *f* stessa, e il cui nome è indicativo della sua caratteristica fondamentale: essere un contenitore NON VUOTO, perché conterrà, ALMENO, l'elemento che ci ha ficcato dentro la funzione *tuple\_cat* invocata in precedenza e dichiarata nel *namespace std*, la quale concatena l'esito di *tie* con la *tuple* contenente gli elementi presenti nel *parameter pack* trasmesso a *f*.
4. Il metodo *f* è, a sua volta, direttamente invocato da *main* per DUE VOLTE, la prima come metodo dell'oggetto *ciccio* e la seconda come metodo dell'oggetto *coccio*, che sono due istanze TOTALMENTE DIVERSE della classe *template Ciccio*. Nei punti successivi si seguirà passo per passo il flusso di questa sezione di programma.
5. Alla prima riga di *main* viene istanziato *ciccio* come oggetto della *class Ciccio<int, int, 4, Peppo, Puppo, double, int, char\*>*, che è un'istanziazione particolare della *template class Ciccio*. Quando il compilatore s'imbatte in questa prima riga, si mette a generare il codice binario relativo a QUELLA istanziazione di *Ciccio*, così che sia possibile realizzare l'oggetto *ciccio* e tutti gli eventuali suoi fratelli omozigoti; durante tale processo, i tipi inseriti nella *lista template* di *Ciccio* assumono i seguenti valori:
   * *A ---------> int*
   * *B ---------> int*    (NON *double*)
   * *Y ---------> Peppo*    (NON *Pappo*)
   * *Z ---------> Puppo*
   * *Altri -----> double, int, char\** , ossia un *parameter pack* contenente, nell'ordine, i tre tipi citati,

e alla variabile intera *N*, che NON È, giovi ricordarlo, una variabile membro, è assegnato il valore 4 (NON 5).

1. L'oggetto *ciccio* è costruito, come si vede, attraverso il costruttore di *default*, la cui lista di inizializzazione, posta dopo il segno di due punti, implica che le variabili membro assumano i seguenti valori:
   * *a* (che è un *int*) ----> 1
   * *b* (che è un *int*) ----> 2
   * *k* (che è un *int*) ----> 4
   * *y* (che è un *Peppo<int>*) ----> costruito col suo costruttore monoparametrico di argomento intero
   * *z* (che è un *Puppo<Pappo>*) ----> costruito col suo costruttore monoparametrico di argomento intero: tale costruttore inizializza, col valore ricevuto, la variabile membro di *z* denominata *x*, la quale, a sua volta, è un *Pappo<double>*.
2. Tutto ciò è confermato da quanto scrive il metodo *info( )* invocato subito dopo l'istanziazione; e siccome il *parameter pack* contiene i tipi che si sono detti, ecco perché il metodo *f*, che riceve GIUSTO quel *pack* come argomento, deve essere invocato con TRE parametri, non uno in più o in meno, di tipi coerenti.
3. DEL TUTTO DIVERSO è l'oggetto *coccio*, che appartiene alla classe *Ciccio<double>*: stavolta la traduzione in binario operata dal compilatore lascia tutti gli elementi della *lista template* ai loro valori *standard* eccettuato il primo che, essendone privo, DEVE ESSERE SEMPRE SPECIFICATO e, nel caso presente, è *double*. Il *parameter pack*, questa volta, è vuoto e questo spiega perché il metodo *f*, se invocato come metodo di *coccio*, NON DEVE RICEVERE ALCUN ARGOMENTO, pur essendo LO STESSO METODO DI PRIMA.
4. Qualcosa di completamente analogo accade anche alla funzione *funza*, che non appartiene ad alcuna classe, ma, come spiega anche la stessa *main*, gode della stessa amplissima flessibilità della classe *Ciccio*, sia rispetto al tipo sia rispetto al numero dei propri argomenti, maggiore o uguale a due.

Ora ci si concentri sul modo di lavorare del metodo *f* delle classi *Ciccio*: il plurale è voluto, trattandosi di istanziazioni di una stessa classe *template*. Esso riceve il numero e il tipo di argomenti che compete al *parameter pack* chiamato *Altri*, vale a dire essenzialmente QUALUNQUE COSA, in numero, tipo e ordine. Tutto il contrario di quello che si attende una funzione normale: un preciso NUMERO di argomenti, di tipo espressamente dichiarato, e sempre nello STESSO ORDINE.  
Nonostante questa **TOTALE ARBITRARIETÀ** la funzione, per come è stata scritta, sa perfettamente riconoscere:

* il numero di argomenti che ha ricevuto: glielo dice l'operatore *sizeof...*, in cui i tre puntini fanno parte dell'operatore;
* il tipo di ciascun argomento, nell'ordine in cui è stato ricevuto, ossia nell'ordine con cui è stato inserito nel *parameter pack* all'atto dell'istanziazione dell'oggetto proprietario, e, naturalmente, il suo valore.

Quest'ultimo risultato è, appunto, il frutto dell'azione combinata degli oggetti *tuple* e *variant*; quando il metodo *f* inizia a eseguirsi, per prima cosa istanzia proprio un oggetto, chiamato *Tupla*, appartenente alla classe *tuple<Altri...>*. L'oggetto è immediatamente inizializzato tramite l'inizializzatore *{quq...}*; in questa riga di codice è materializzato il concetto di espansione di un *parameter pack*: la sintassi che fa uso dei tre puntini, senza niente alla loro destra, implica appunto l'espansione di ciò che si trova alla loro sinistra secondo quanto è stato inserito al momento dell'istanziazione. A tutti i fini pratici, quando *f* è invocata da parte dell'oggetto *ciccio* di *main*, la sua prima riga è come se fosse scritta

*tuple<double, int, char\*>  
Tupla{1.4, 1, const\_cast<char \*>("cucu")};*

vale a dire con la normale sintassi di un'istanziazione di una classe *template* e con l'altrettanto normale sintassi di un inizializzatore tra parentesi graffe. L'oggetto *Tupla* finisce con l'essere una sorta di pseudo*array* di tre elementi, ognuno, nell'ordine, del tipo elencato nella *lista template* di *tuple*: l'operatore *const\_cast*, citato nel canto 12 e su cui si tornerà, è fortemente consigliato per rendere compatibile la costante *"cucu"* col tipo *char \**, privo del qualificatore *const*.  
  
La STESSA prima riga della funzione *f*, quando eseguita attraverso l'oggetto *coccio* di *main*, è interpretata dal compilatore come se fosse scritta

*tuple< > Tupla{ };*

generando una *tuple* vuota.  
  
Proseguendo l'analisi di *f* si incontra la dichiarazione dell'oggetto *Tupla\_con\_almeno\_un\_elemento*, in cui va apprezzato l'uso della parola di vocabolario *auto* che lascia al compilatore il compito di stabilirne il tipo: il compilatore se la cava benissimo, riconoscendo facilmente che il tipo di *Tupla\_con\_almeno\_un\_elemento* è quello restituito dalla funzione *tuple\_cat*, ossia un'ulteriore istanza della classe *tuple* della cui *lista di templatizzazione* il programmatore non ha quindi alcun bisogno di occuparsi.  
  
La funzione *tuple\_cat* svolge l'interessante ufficio di concatenare, nell'ordine, in un'unica *tuple* le due *tuple* che le sono trasmesse come parametri; la *tuple* così concatenata sarà, per l'appunto, quella restituita, che inizializzerà *Tupla\_con\_almeno\_un\_elemento*. Va da sé, quindi, che *tuple\_cat* è, PER FORZA, dichiarata anch'essa come una funzione *template*, così come la funzione *tie* che, a sua volta, crea una *tuple* col materiale che le viene trasmesso: in questo caso il solo intero *n*. L'effetto finale, come dovrebbe ormai essersi intuito, è che *Tupla\_con\_almeno\_un\_elemento* risulta essere uno pseudo*array* con UN ELEMENTO IN PIÙ rispetto a *Tupla*, posto in prima posizione; pertanto *Tupla\_con\_almeno\_un\_elemento*, come autocommenta il suo stesso nome, NON È MAI una *tuple* vuota. Ciò è essenziale in vista della trasmissione alla funzione *dget* che avviene appresso, subito dopo il controllo compiuto sul *parameter pack* tramite l'operatore *sizeof...*, che consente la scrittura della frase *non ho ricevuto ALCUN ARGOMENTO* qualora *sizeof...(Altri)* restituisca zero.  
  
Si noti che la funzione *dget* viene eseguita, sotto controllo di *try*, dall'interno di un ciclo, potenzialmente infinito, numerato col contatore *i* che parte dal valore 1 e si incrementa a ogni iterazione del ciclo: in sostanza si scorre il contenuto di *Tupla\_con\_almeno\_un\_elemento* a partire dalla sua seconda posizione; la prima non interessa: è quello che ci ha messo *tuple\_cat*. Ciò che *dget* restituisce va a inizializzare l'oggetto *t*, con un'inizializzazione DIVERSA a ogni iterazione; ecco perché la dichiarazione di *t* si giova ancora una volta della parola *auto*: il programmatore **NON AVREBBE POTUTO**, CON TUTTA LA SUA PERIZIA E/O BUONA VOLONTÀ, attribuire a *t* un tipo esplicito (O SOMMA GRANDEZZA del linguaggio, che consente di dichiarare variabili senza neppure sapere che cosa siano...).  
  
Si esamini ora dettagliatamente CHE COSA FA *dget*: visto che il contatore del ciclo si incrementa, prima o poi *dget* finirà per essere invocata con un valore di *i* eccessivo rispetto al contenuto di *Tupla\_con\_almeno\_un\_elemento*; finché questo non accade *i* ha un valore tale che in *Tupla\_con\_almeno\_un\_elemento* qualcosa, nella posizione *i*-esima, C'È.  
Ciò assodato, entrando nell'ambito di *dget*, si osserva che restituisce null'altro che ciò che le fa ricevere *vget<0>* cui trasmette, a sua volta, null'altro che ciò che ha ricevuto da *main*: si usa dire che *dget* è un involucro di *vget<0>*, utile solo per il fatto di nascondere al compilatore la ricorsività di *vget*, in modo tale che, dal punto di vista del compilatore, risulti invocata SOLO con la costante 0 racchiusa tra le parentesi angolari. Questo è essenziale quando si tratta di accedere ai singoli elementi di una *tuple*, dato che l'indice di accesso deve poter essere NOTO al compilatore DURANTE la compilazione, vale a dire che dev'essere una COSTANTE o, al massimo, una *constexpr*; ma su questo si può soprassedere.  
Qui però si pretenderebbe di accedere al contenuto di una *tuple* tramite il contatore di un ciclo (la qual cosa, peraltro, sarebbe evidentemente quella più utile da farsi) il quale è, PER NATURA, tutt'altro che una costante. Ecco da dove nasce la necessità del trucco di ricorrere a un involucro: noi stiamo dicendo al compilatore che ci interessa accedere SEMPRE all'elemento che occupa la posizione COSTANTE zero; al resto pensa la classe *variant*, ricopiando ogni volta, grazie alla ricorsività di *vget* che si attua nella sua ultima riga, nella posizione zero del suo oggetto, restituito al chiamante, l'opportuno elemento dell'oggetto *tuple* ricevuto (ENORME LINGUAGGIO.....).   
  
Dovrebbe essere superfluo sottolineare che dal ciclo si esce non appena *vget* esegue l'operatore *throw* sulla stringa *"Tupla esaurita.\n\n"*, la quale viene inviata sullo *standard output* da *catch(const char \* c)* prima del *break* di interruzione del ciclo; e questo accade appunto allorché risulta vera l'espressione  
*n == sizeof...(T) - 1*  
ossia quando, O subito O per ricorsione, viene eseguita *vget<sizeof...(T) - 1>*.   
  
**Puntualizzazioni, pignolerie e varianti**

1. si è affermato che il metodo *f* sa riconoscere il tipo degli elementi del *parameter pack* che riceve; questo è concettualmente vero, ma non nella presente implementazione, dato che il tipo di *t*, restituito da *vget<0>* è sempre quello di un oggetto della classe *variant<T...>*, indipendentemente dal tipo che vi si trova contenuto. Per ricavare quest'ultimo occorre un approccio del genere di quello adottato nella pagina che descrive dettagliatamente le *variadic template*, il che non era appropriato nel momento attuale del percorso; oppure usare la classe *boost::any*, il che era ancora peggio.
2. le funzioni *template* per i diversi *overload* dell'*operator<<* non sono un'oziosità: non era possibile scriverne una sola con la dichiarazione  
   *template <typename A> ostream& operator<<(ostream &, A);*   
   perché QUESTA sarebbe valsa anche per qualsiasi altro tipo, compresi quelli NATIVI, generando pertanto ambiguità con quelle già definite nel documento incluso *iostream*: provare per credere.
3. le stesse funzioni di cui al punto precedente non hanno bisogno di essere dichiarate *friend* di nessuno, a causa della scelta di lavorare con delle *struct* piuttosto che con delle *class*. Si ricorda (*repetita iuvant*) che comunque la dichiarazione *friend* non è mai necessaria SE NON SI PRETENDE ACCESSO a membri non pubblici di una classe.
4. quando nelle *liste di templatizzazione* appare un *template-template parameter* l'ultima parola di vocabolario da inserirvi è SEMPRE *class*, NON *typename* e men che meno *struct*; in altre parole le seguenti dichiarazioni sono tutte ERRATE, con l'errore evidenziato in grassetto:
   * *template < template<typename>* ***typename*** *A > void funz(A<int>);*
   * *template < template<class>* ***typename*** *A > void funz(A<float>);*
   * *template < template<typename>* ***struct*** *A > void funz(A<double>);*
   * *template < template<class>* ***struct*** *A > void funz(A<char>);*

le uniche GIUSTE essendo, con qualsiasi tipo esplicito diverso da quelli proposti:

* + *template < template<typename> class A > void funz(A<int>);*
  + *template < template<class> class A > void funz(A<long double>);*

1. all'inizio del programma si trovano DUE linee *using namespace*: in effetti il pacchetto *boost* effettua le proprie dichiarazioni entro un proprio, omonimo, *namespace*. Tanto per esemplificare, la *template* denominata *variant* avrebbe il nome completo *boost :: variant*, da usarsi se si toglie la linea *using namespace boost;*. Invece le funzioni che concernono le *tuple*, e la classe medesima, sono definite nel *namespace std* e quindi andrebbero prefissate, in assenza della *using namespace std;*, col consueto risolutore di ambito *std ::*
2. qualche parola in più va spesa a proposito della cosiddetta espansione di un *parameter pack*; già qualcosa dovrebbe essere stato intuito all'esame del programma e a un'attenta lettura delle spiegazioni fin qui fornite, ma è opportuno, a questo punto, dettare la regola generale, che recita quanto segue: **il nome di un *parameter pack*, o quello rappresentativo degli elementi che contiene, seguito da tre puntini comporta, in ogni contesto in cui appaia lecitamente, la sostituzione dell'intero contesto con una lista di termini separati da virgole, ognuno dei quali è una mera sostituzione, nell'ordine di apparizione nel *pack*, di uno dei suoi elementi in una replica dell'intero contesto**.  
   Resta da stabilire quali siano i contesti in cui il costrutto *nome\_pack...* appaia lecitamente, come richiede la regola: li potete trovare, ampiamente discussi ed esemplificati, nel documento intitolato *"pack expansion"*, raggiungibile dall'omonimo collegamento posto nelle regole generali del C++.

Nel programma proposto all'inizio di questo canto si trova definita una specializzazione di una classe *template* che, allo stato dell'arte, NON VIENE USATA. Provate a individuarla e ad apportare a *main* qualche aggiunta e/o cambiamento che la induca a farne uso; provate altresì ad aggiungere altre istanze della classe *Ciccio*, a vostro talento, e/o altre specializzazioni di classi esistenti o addirittura NUOVE classi *template* ideate da voi. Prendete questo invito come una salutare pausa di riflessione.

**Canto trentesimo: dopo le virtù del programmatore...le virtualità del linguaggio; parte I: il problema del diamante.**

Considerate la seguente situazione, in cui alcune classi si trovano fra loro in rapporti di ereditarietà, come già visto nei canti 24 e 25:

# include <iostream>  
using namespace std;  
  
class Prima  
{// omissis  
protected:  
int prima;  
public:  
Prima(int i) {prima = i;}  
Prima( ) {prima = 33;}  
};  
  
class Seconda : public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda(int i) : Prima(i) {seconda = i\*2;}  
Seconda( ) {seconda = 66;}  
};  
  
class Terza : public Seconda {// omissis  
protected:  
int terza;  
public:  
Terza(int i) : Seconda(i) {terza = i\*3;}  
Terza( ) {terza = 99;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< "; terza = " << terza << '\n';}  
};  
  
int main( )  
{  
Terza oggetto, oggetto\_(2);  
cout << "valori di oggetto:\n",  
oggetto . scrivi\_tutto( ),  
cout << "valori di oggetto\_:\n",  
oggetto\_ . scrivi\_tutto( );  
}

L'esecuzione di questo programma (**ESEGUITELO!**) non provoca alcuna sorpresa (avrebbe dovuto?).  
Tuttavia, provate a eseguire la seguente variante:

# include <iostream>  
using namespace std;  
  
class Prima  
{// omissis  
protected:  
int prima;  
public:  
Prima(int i) {prima = i;}  
Prima( ) {prima = 33;}  
};  
  
class Seconda : public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda(int i) : Prima(i) {seconda = i\*2;}  
Seconda( ) {seconda = 66;}  
};  
  
class Seconda\_ : public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda\_(int i) : Prima(i) {seconda = -i\*2;}  
Seconda\_( ) {seconda = -66;}  
};  
  
class Terza : public Seconda, public Seconda\_  
{// omissis  
protected:  
int terza;  
public:  
Terza(int i) : Seconda(i), Seconda\_(i) {terza = i\*3;}  
Terza( ) {terza = 99;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< "; terza = " << terza << '\n';}  
};  
  
int main( )  
{  
Terza oggetto, oggetto\_(2);  
cout << "valori di oggetto:\n",  
oggetto . scrivi\_tutto( ),  
cout << "valori di oggetto\_:\n",  
oggetto\_ . scrivi\_tutto( );  
}

in cui è stata aggiunta una seconda classe *Seconda\_*; praticamente un clone della precedente: si perdoni il bisticcio verbale, ma lo si è quasi cercato... e si fa ereditare *Terza* da entrambe le classi *"Seconde"*.  
  
**SORPRESA(!)** (ma neanche tanto): questo programma NON SI COMPILA.  
  
E, quel che è peggio, dei due errori che vengono riscontrati il secondo è comprensibile e lo stesso suggerimento dato dal compilatore per rimediarlo è chiaro e quasi tale da mettere a disagio il povero programmatore per non averci pensato da sé...ma il PRIMO errore che diamine significa? E che razza di suggerimento è quello che proviene dal compilatore?  
  
Siccome vi conosco, e so che ci sarà sempre qualcuno che crede di essere più furbo degli altri e NON OBBEDISCE agli ordini che impongono di eseguire i programmi, per costoro, ma anche per gli altri, riporto qui di seguito i diagnostici del compilatore GNU 4.8.3:

In member function 'void Terza::scrivi\_tutto( )':  
  
:40:19: error: reference to 'prima' is ambiguous  
<< "prima = " << prima  
  
:7:6: note: candidates are: int Prima::prima  
int prima;  
  
:7:6: note: int Prima::prima  
  
:41:23: error: reference to 'seconda' is ambiguous  
<< "; seconda = " << seconda  
  
:25:6: note: candidates are: int Seconda\_::seconda  
int seconda;  
  
:16:6: note: int Seconda::seconda  
int seconda;

Ora, come si era detto, si può comprendere che il compilatore giudichi ambiguo citare una variabile chiamata *seconda* in un metodo della classe *Terza*, perché di variabili con tale nome un oggetto della classe *Terza* ne possiede veramente DUE, una che gli proviene in eredità dalla classe *Seconda* e l'altra dalla classe *Seconda\_*...e in effetti il compilatore ce le indica entrambe come candidate a risolvere l'ambiguità; sta al programmatore essere CHIARO sulle proprie intenzioni: se intende scriverne una sola deve citarla completa del proprio risolutore di ambito; se invece intende farle scrivere entrambe...deve ugualmente citare ciascuna col SUO risolutore di ambito.  
  
Ma perché il compilatore si lagna anche riguardo alla variabile chiamata *prima*? Solo la classe *Prima* ha una variabile con quel nome: perché mai dovrebbe sorgere un'ambiguità? È chiaro che quando scrivo *prima* (qui è lo *scriptor inops* che parla in prima persona, NON l'autore) intendo QUELLA VARIABILE LÌ!  
  
...Se non che la classe *Terza* NON eredita dalla classe *Prima* DIRETTAMENTE, ma per il tramite delle DUE classi *"Seconde"*: sono LORO a essere eredi DIRETTE di *Prima* e quindi ENTRAMBE, in maniera del tutto indipendente l'una dall'altra, si ritrovano nel corredino la variabile *prima*.  
  
Quando poi diventano ENTRAMBE antenate DIRETTE di *Terza*, ENTRAMBE trasmettono alla propria erede, in modo INDIPENDENTE l'una dall'altra, **TUTTO** ciò che sono legittimate a trasmetterle e quindi ANCHE la variabile *prima* ereditata dall'antenata comune.  
  
Ecco perché un oggetto della classe *Terza* si ritrova con DUE variabili membro chiamate tutte e due *prima* e collocate in due indirizzi di memoria distinti, dove sono state poste INDIPENDENTEMENTE all'atto dell'istanziazione dell'oggetto.  
  
Da qui nasce l'ambiguità, perché il compilatore associa SEMPRE i nomi agli indirizzi in memoria e qui ci sono DUE indirizzi distinti etichettati con lo stesso nome, esattamente come se avessimo apposto due etichette IDENTICHE su due diversi cassetti di un classificatore (la memoria); a quel punto la frase *"dimmi che cosa c'è nel cassetto con l'etichetta PencaPolla"* potrebbe avere come SOLA risposta sensata la domanda *"quale? Ce ne sono due!"*.  
  
Ma il peggio non è ancora arrivato, perché, in un caso simile, NEMMENO IL RISOLUTORE DI AMBITO risolve un bel NULLA: basta leggere, per l'appunto, il diagnostico emesso dal compilatore, secondo il quale l'ambiguità sarebbe risolvibile da una tra DUE possibili candidate chiamate...ENTRAMBE *Prima::prima*. Ed è del tutto ovvio che si chiamino tutt'e due così, vista la loro provenienza: le etichette sui cassetti sono SUL SERIO IDENTICHE, in tutto e per tutto.  
  
Quest'apparentemente insormontabile *impasse* va sotto il nome di problema del diamante perché il grafo della gerarchia ereditaria di cui si sta discutendo ha, evidentemente, la forma di una losanga e il simbolo della losanga viene anche detto *diamond* in angloamericano (solo loro potevano chiamare diamante un rombo...).  
  
Che si tratti di un problema penso sia ormai chiaro, anche se taluno potrebbe obiettare: *"ma perché porsi in una simile situazione?"* La risposta più banale potrebbe essere *"e perché no?"* ma per essere un po' meno ovvio dirò che non si vede un valido motivo per ritenere illecita una gerarchia di questo genere: dopo tutto, nella vita reale, NESSUNO ha mai sofferto di problemi genetici per il fatto di avere o non avere degli zii.  
  
Qualcun altro potrebbe chiedersi per quale ragione il compilatore faccia tanto il sofistico: in fondo, anche se ci fossero due cassetti con la stessa etichetta, non basterebbe aprirne uno qualsiasi?  
Per ribattere a questa obiezione viene ancora in soccorso il Divino Poeta, quando scrisse:

Ed elli a me "Perché tanto delira",  
disse, "lo 'ngegno tuo da quel che sòle?  
o ver la mente dove altrove mira?"

In primo luogo non è assolutamente detto che il contenuto dei due cassetti sia lo stesso: dipenderebbe dalla catena dei costruttori eseguiti nelle linee ereditarie *Prima--->Seconda--->Terza* e *Prima--->Seconda\_--->Terza*; il compilatore non potrebbe assumersi la responsabilità di aprire un cassetto a proprio piacere.  
  
In secondo luogo, anche ammettendo (e NON è così) che il contenuto dei cassetti fosse SEMPRE identico, come le etichette, CHI MAI PUÒ ESSERE tanto PIRLA da tollerare che sia sprecata memoria in ragione del 100% dello spazio occupato dalle variabili che *Terza* eredita da *Prima*?  
  
E, in definitiva, un linguaggio che voglia chiamarsi GRANDE non può inciampare miseramente su un problema di questo genere: è una questione d'onore...  
Ecco perciò la versione del programma che RISOLVE il problema:

# include <iostream>  
using namespace std;  
  
class Prima  
{// omissis  
protected:  
int prima;  
public:  
Prima(int i) {prima = i;}  
Prima( ) {prima = 33;}  
};  
  
class Seconda : public virtual Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda(int i) : Prima(i) {seconda = i\*2;}  
Seconda( ) {seconda = 66;}  
};  
  
class Seconda\_ : virtual public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda\_(int i) : Prima(i) {seconda = -i\*2;}  
Seconda\_( ) {seconda = -66;}  
};  
  
class Terza : public Seconda, public Seconda\_  
{// omissis  
protected:  
int terza;  
public:  
Terza(int i) : Seconda(i), Seconda\_(i) {terza = i\*3;}  
Terza( ) {terza = 99;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << Seconda::seconda  
<< "; seconda = " << Seconda\_::seconda  
<< "; terza = " << terza << '\n';}  
};  
  
int main( )  
{  
Terza oggetto, oggetto\_(2);  
cout << "valori di oggetto:\n",  
oggetto . scrivi\_tutto( ),  
cout << "valori di oggetto\_:\n",  
oggetto\_ . scrivi\_tutto( );  
}

Come si potrà apprezzare per confronto diretto, è bastato aggiungere la parola di vocabolario *virtual* alla clausola che specifica la figliolanza delle due classi *"Seconde"*, in un ordine indifferente rispetto alla parola *public*.  
  
In sostanza, con questa informazione, si avverte il compilatore che, qualora le classi *Seconda* e *Seconda\_* diventassero insieme antenate di un'altra classe, come poi di fatto accade, l'eredità proveniente da *Prima* non dovrà essere trasmessa con la destra che non sa quello che fa la sinistra, ma di concerto, in modo che la classe erede erediti da *Prima* quello che le compete UNA SOLA VOLTA.  
  
A questo punto la LOGICA suggerisce che la SOLA variabile *prima* ereditata dagli oggetti della classe *Terza* non può essere NESSUNA delle due variabili *prima* possedute per eredità dalle due antenate dirette che, come si è visto, potrebbero essere diverse: quale andrebbe trasmessa? Ma dovrà essere una variabile *prima* che gli oggetti della classe *Terza* si scelgono autonomamente all'atto della loro istanziazione, vale a dire che è il PROGRAMMATORE che la sceglie: GRAN LINGUAGGIO, MA GRANDE DAVVERO.  
  
*"Come avverrà questo?"* dovreste chiedere voi, imitando Chi so io...  
  
Se foste stati ligi ai comandi, e aveste eseguito l'ultima versione del programma, vi DOVRESTE essere accorti che il valore della variabile *prima* è UGUALE per i due oggetti istanziati, diversamente da quanto avveniva nella versione primigenia, quella in cui la *class Seconda\_* neppure esisteva. E tale valore comune è quello impostato dal costruttore di *default* della classe *Prima*: ve ne eravate accorti?  
  
Ciò può significare una sola cosa: che, quantunque *oggetto\_* sia stato costruito tramite il costruttore parametrico di *Terza*, e quindi questi abbia invocato, dalla propria lista di inizializzazione, gli omologhi costruttori di *Seconda* e *Seconda\_*, NESSUNO DI QUESTI ULTIMI DUE, come del resto si era anticipato, si è azzardato a invocare il costruttore parametrico di *Prima*, nonostante esso sia ben presente nella lista di inizializzazione di entrambi, e pertanto l'oggetto della classe *Prima* che è parte di *oggetto\_* è stato costruito utilizzando il costruttore *default* di *Prima*, così come è apparso evidente dal semplice esame dello *standard output* del programma.   
  
E qui viene il bello: in deroga alla regola generale (rileggetevi CON ATTENZIONE il canto 24, se non l'avete presente), secondo cui un costruttore può invocare, dalla propria lista di inizializzazione, SOLAMENTE costruttori delle classi ANTENATE DIRETTE, come fa il costruttore di *Terza* coi costruttori delle due classi *"Seconde"*, quando, nell'albero genealogico della gerarchia ereditaria, una classe viene ereditata con la qualifica *virtual*, i suoi costruttori possono essere invocati ANCHE dalle liste di inizializzazione dei costruttori degli eredi più lontani.  
  
Ecco pertanto *"come avverrà"*: sarà sufficiente inserire direttamente l'invocazione del costruttore di *Prima* che si preferisce nella lista di inizializzazione del costruttore parametrico di *Terza*, facendola diventare come in:

*Terza(int i) :   
Prima(i), Seconda(i), Seconda\_(i) {terza = i\*3;}*

Provate e vedrete; quale altro aggettivo usare per un simile linguaggio? GRANDE pare ormai insufficiente...  
  
Per darvene ulteriore dimostrazione, la deroga alla regola varrebbe anche per un'eventuale classe *Quarta* che fosse erede di *Terza* e così via fino a un'erede *Ennesima* che si inserisse nella STESSA linea ereditaria dopo innumerevoli generazioni: anche QUEST'ULTIMA erede godrebbe del diritto di invocare i costruttori di *Prima* DIRETTAMENTE dalla lista d'inizializzazione dei propri costruttori.  
  
Ma allora...  
  
Càspita, dove sta scritto che la parola *virtual* debba essere usata SOLAMENTE PER RISOLVERE IL PROBLEMA DEL DIAMANTE? Non è, per caso, che la soluzione di quel problema sia un semplice sottoprodotto?  
  
Se fosse vero quello che DOVREBBE ESSERE VENUTO IN MENTE A CHIUNQUE SIA SVEGLIO... se uno avesse bisogno di invocare un costruttore che NON SIA di un'antenata diretta...basterebbe appellarsi alla deroga e appiccicare la qualifica *virtual* nella specifica ereditaria della prima erede della classe il cui costruttore occorrerebbe invocare? È così?...  
  
**È COSÌ!**  
  
...senza contare che, a partire dallo *standard* 2011 del linguaggio, i COSTRUTTORI STESSI possono perfino essere ereditati...  
  
INEFFABILE LINGUAGGIO...

**Canto trentunesimo: dopo le virtù del programmatore...le virtualità del linguaggio; parte II: i metodi virtuali**

*O sol che sani ogne vista turbata,  
tu mi contenti sì quando tu solvi,  
che, non men che saver, dubbiar m'aggrata.*   
  
Se vi trovate in questo auspicabile e lodevole stato d'animo, ecco pronta per voi una nuova sfida che vi gratificherà assai, permettendovi di molto *dubbiare*, sì che, quando vi siano sciolti i nodi della conoscenza, siate contenti quasi fino all'appagamento.  
  
Si utilizzerà ancora la gerarchia ereditaria del canto precedente, ma in una versione accorciata e leggermente modificata, ossia:

class Prima  
{// omissis  
protected:  
int prima;  
public:  
Prima(int i) {prima = i;}  
Prima( ) {prima = 33;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< '\n';}  
};  
  
class Seconda : public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda(int i) : Prima(i) {seconda = i\*2;}  
Seconda( ) {seconda = 66;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< '\n';}  
};  
  
class Terza : public Seconda  
{// omissis  
protected:  
int terza;  
public:  
Terza(int i) : Seconda(i) {terza = i\*3;}  
Terza( ) {terza = 99;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< "; terza = " << terza << '\n';}  
};

Niente di particolarmente sconvolgente: una gerarchia liscia e piana... che parrebbe immobile se non fosse il tremolare e l'ondeggiar leggero della luna... ma questa è un'altra storia; intendo una gerarchia che non presenta problemi di diamanti o altre pietre preziose, quali eredità virtuali e costruttori eseguiti da lunge.  
  
L'unica cosa da notare, se proprio si volesse notare qualcosa, sarebbe che ciascuna classe ha un metodo chiamato *void scrivi\_tutto( )*, di modo che la classe ultima erede finisce per possederne tre versioni: la sua propria, quella che le proviene in eredità dall'antenata diretta e quella che le perviene dall'antenata lontana per il tramite dell'antenata diretta; quest'ultima, a sua volta e per l'identico meccanismo, ne possiede due, mentre ovviamente la classe capostipite ne ha una sola.  
  
Nel caso presente non si genera alcuna ambiguità, poiché la richiesta di esecuzione di *scrivi\_tutto( )* SENZA SPECIFICARE ALCUN RISOLUTORE DI AMBITO sarà risolta facendo eseguire la versione che è proprietà PERSONALE, non ereditata, dell'oggetto che compie la richiesta.  
In altre parole una funzione *main* così scritta:

int main( )  
{  
Prima p(1);  
Seconda s(2);  
Terza t(3);  
p . scrivi\_tutto( ),  
cout << "\_\_\_\_\_\_\_\n\n",  
s . Prima :: scrivi\_tutto( ),  
s . scrivi\_tutto( ),  
cout << "\_\_\_\_\_\_\_\n\n",  
t . Prima :: scrivi\_tutto( ),  
t . Seconda :: scrivi\_tutto( ),  
t . scrivi\_tutto( );  
}

non riserverà alcuna sorpresa e produrrà, una volta eseguito il codice, il seguente, DEL TUTTO ATTESO (o no?), *standard output*:

prima = 1  
\_\_\_\_\_\_\_  
  
prima = 2  
prima = 2; seconda = 4  
\_\_\_\_\_\_\_  
  
prima = 3  
prima = 3; seconda = 6  
prima = 3; seconda = 6; terza = 9

Come dite? Non vi è venuto così? ALLORA RICOMINCIATE DAL CANTO ZERO.  
  
Quelli che sono rimasti sono invitati a modificare *main* cambiando le dichiarazioni degli oggetti in dichiarazioni di puntatori, ossia:  
  
*Prima p(1) ------------> Prima \*p = new Prima(1)  
Seconda s(2) ------------> Seconda \*s = new Seconda(2)  
Terza t(3) ------------> Terza \*t = new Terza(3)*  
**OVVIAMENTE**, quando si andrà a richiedere l'esecuzione delle varie *scrivi\_tutto*, l'operatore . (punto) **dovrà, COERENTEMENTE, essere sostituito OVUNQUE con l'operatore -> (freccia)**.   
  
Tutto ciò compiuto, e rieseguito il programma, si constaterà, SENZA VERUNA SORPRESA, nulla essere cambiato.  
  
Supponiamo però che, durante la modifica compiuta sulle dichiarazioni, si sia abusato del copia-incolla, fonte di ogni nequizia, di modo che, dimenticando di apportare TUTTE le correzioni dovute alle linee copincollate partendo da   
  
*Terza \*t = new Terza(3);*   
  
si finisca per avere le tre dichiarazioni seguenti:

Terza \*p = new Prima(1);  
Terza \*s = new Seconda(2);  
Terza \*t = new Terza(3);

(dimenticando, in sostanza, la sostituzione del primo *Terza*)  
  
State *dubbiando* abbastanza?  
  
Non ve n'è ragione: il programma NON SI COMPILA a causa di errori IDENTICI riscontrati nelle prime due dichiarazioni. Il compilatore si lagnerà aspramente, parlando di *invalid conversion from Prima\* to Terza\** (o anche *from Seconda\* to Terza\**). Come dargli torto? Imparate a minimizzare l'uso del copia-incolla e comunque a correggere PER INTERO le linee incollate!   
  
Tuttavia lo STESSO ERRORE di copincollaggio si sarebbe potuto commettere se si fosse partiti dalla prima dichiarazione, giungendo alla situazione complementare in cui le tre dichiarazioni risonino così:

Prima \*p = new Prima(1);  
Prima \*s = new Seconda(2);  
Prima \*t = new Terza(3);

Ebbene...  
  
se date in pasto al compilatore QUESTO codice (ovviamente assieme al resto di *main* e alle tre definizioni delle classi), esso non batterà ciglio sulle dichiarazioni (nessuna *invalid conversion* lamentata [?!?!]) e vi segnalerà UN SOLO ERRORE, **ALTROVE**, con la risibile motivazione *"Seconda is not a base of Prima"*, il che, tradotto in italiano, significa che *Seconda* non è un'antenata di *Prima*. Verrebbe quasi da rispondergli *"Grazie al CENSURA"*...  
  
UN MILIONESIMO DI TRENTESIMO GRATIS ALL'ESAME A CHI INDOVINA CHE COSA STIA SUCCEDENDO PRIMA DI PROCEDERE NELLA LETTURA.  
  
Basta eseguire l'analisi logica della dichiarazione   
  
*Prima \*t = new Terza(3);*  
  
(per quella precedente il ragionamento è identico); l'espressione *new Terza(3);* ha per valore, come si DOVREBBE già sapere, quello restituito dall'operatore *new* ossia un puntatore a un oggetto innominato appartenente alla classe *Terza*, che viene contestualmente costruito dal costruttore parametrico di tale classe, cui viene trasmesso il parametro intero costante 3.  
  
Tale puntatore, trovandosi a destra dell'operatore di assegnamento, ne costituisce l'operando destro e viene quindi assegnato all'operando sinistro.  
Quest'ultimo, però, è un puntatore alla classe *Prima* in corso di dichiarazione e quindi prenderebbe il valore dell'operando destro come propria immediata inizializzazione.  
  
Appare quindi EVIDENTEMENTE necessaria la conversione di un puntatore a *Terza* in un puntatore a *Prima*: è ciò possibile in maniera ben definita? Certo che sì, perché l'oggetto della classe *Terza* puntato dall'operando destro contiene, **PER NATURA**, al proprio interno un oggetto della classe *Prima*, il quale, anzi, è stato addirittura costruito per primo.  
  
Pertanto il dichiarando puntatore *Prima \*t* viene correttamente e senza alcuna ambiguità inizializzato in modo da puntare QUELL'OGGETTO DELLA SUA CLASSE che sta DENTRO l'oggetto innominato, della classe *Terza*, puntato dall'operando destro dell'operatore di assegnamento (GRAN LINGUAGGIO).   
  
La conversione inversa produce errore, come si è visto, perché un oggetto della classe *Prima* NON HA DENTRO alcun altro oggetto da far puntare a puntatori delle classi eredi (è così ovvio da apparire quasi disarmante).  
  
A questo punto però, se *t* punta, come DEVE, un oggetto della classe *Prima*, un tale oggetto, come si sa, possiede UNA SOLA VERSIONE del metodo *void scrivi\_tutto( )*, perché non gliene provengono altre in alcun modo, e di quest'UNICA versione si può invocare l'esecuzione O citandola col solo nome O col risolutore d'ambito (superfluo) *Prima ::*; **CERTO NON** col risolutore d'ambito *Seconda ::*, perché *"Seconda is not a base of Prima"*, come era OVVIO fin dal principio e come dovrebbe essere chiaro, ADESSO, il motivo della segnalazione d'errore alla linea  
  
*t -> Seconda :: scrivi\_tutto( ),*  
  
Commentando questa linea il programma si compila perfettamente, e la sua esecuzione dà una conferma evidente di quanto fin qui detto, al solo esame dello *standard output* prodotto, che qui si riporta a vantaggio degli ignavi (si spera non ve ne sia alcuno fra voi).

prima = 1  
\_\_\_\_\_\_\_  
  
prima = 2  
prima = 2  
\_\_\_\_\_\_\_  
  
prima = 3  
prima = 3

Questo canto, fin qui, non ha detto nulla di particolarmente interessante; quel che si è visto, tutto sommato, non era neppure troppo celato tra le pieghe di ciò che già si sarebbe dovuto sapere o intuire circa i comportamenti fini del compilatore in base alle regole del linguaggio. Soprattutto, finora, non è stato fatto alcun cenno alla virtualità di cui si fa menzione nel titolo del canto.  
  
È il momento di arrivare al sodo: prendete l'ultima versione del programma che avete a disposizione (per intendersi: quella che ha prodotto lo *standard output* qui sopra) e **aggiungete SOLAMENTE, davanti alla dichiarazione/definizione di *void scrivi\_tutto( )* della classe *Prima***, la parola di vocabolario *virtual*. Per comodità di consultazione si riporta qui appresso il programma come si vuole che sia, con evidenziata in grassetto **la SOLA differenza** con la versione che ha generato il risultato di sopra e di cui **così bene si è compreso il comportamento e il significato**:

# include <iostream>  
using namespace std;  
  
class Prima  
{// omissis  
protected:  
int prima;  
public:  
Prima(int i) {prima = i;}  
Prima( ) {prima = 33;}  
**virtual** void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< '\n';}  
};  
  
class Seconda : public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda(int i) : Prima(i) {seconda = i\*2;}  
Seconda( ) {seconda = 66;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< '\n';}  
};  
  
class Terza : public Seconda  
{// omissis  
protected:  
int terza;  
public:  
Terza(int i) : Seconda(i) {terza = i\*3;}  
Terza( ) {terza = 99;}  
void scrivi\_tutto( )  
{cout  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< "; terza = " << terza << '\n';}  
};  
  
int main( )  
{  
Prima \*p = new Prima(1);  
Prima \*s = new Seconda(2);  
Prima \*t = new Terza(3);  
p -> scrivi\_tutto( ),  
cout << "\_\_\_\_\_\_\_\n\n",  
s -> Prima :: scrivi\_tutto( ),  
s -> scrivi\_tutto( ),  
cout << "\_\_\_\_\_\_\_\n\n",  
t -> Prima :: scrivi\_tutto( ),  
t -> scrivi\_tutto( );  
}

**ESEGUITE QUEST'ULTIMA VERSIONE DEL PROGRAMMA...**  
  
**TOMBOLA!**  
  
CINQUE MILIONESIMI DI TRENTESIMO A CHI SA SPIEGARE QUEL CHE È SUCCESSO (senza leggere oltre questa linea, OVVIAMENTE).  
  
Il metodo di indagine che si seguirà per svelare il mistero sarà degno del migliore Sherlock Holmes; si parte dalla raccolta degli indizi che emergono dall'osservazione sperimentale del fatto. Sono inconfutabili le seguenti evidenze:

1. dall'esame dello *standard output* non risulta più equivalente richiedere l'esecuzione di *scrivi\_tutto( )* senza o con il risolutore di ambito esplicitamente indicato;
2. tra i due programmi che hanno dato risultati tanto diversi sussiste UNA SOLA DIFFERENZA: appunto l'inserimento di *virtual* là dove è stata scritta;
3. come corollario del punto precedente, *main* non è stata AFFATTO modificata: sono stati istanziati gli stessi oggetti, usati gli stessi costruttori, a loro volta INTONSI, e dichiarati e inizializzati in ugual maniera gli STESSI puntatori. Inoltre sono state effettuate le STESSE richieste di esecuzione di metodi utilizzando la stessa sintassi.

*"Elementare, Watson!"* - direbbe a questo punto l'uomo di Baker Street - *"La responsabile di un'azione così geniale non può essere che Lei, la Donna; non so ancora come abbia fatto, ma lo scoprirò con un semplice esperimento!"*  
  
In realtà l'ottimo Sherlock sta prendendo una cantonata, indotta dalla sua sconfinata ammirazione per Irene Adler; a meno che *virtual* non sia un suo alias cifrato, questa volta Lei non c'entra, anche perché, al tempo in cui si suppone che visse, il C++ non era stato ancora formulato.  
  
È pur vero, tuttavia, che un *"semplice esperimento"* possa aiutare a comprendere come *virtual* sia riuscita a operare quel che ha fatto: provate a sostituire la dichiarazione del puntatore *t* con la seguente  
 *Seconda \*t = new Terza(3);*   
  
e rieseguite il programma così modificato.  
  
**COME NON FOSSE STATO TOCCATO NULLA...**   
  
**??????**  
  
*"Mio caro Watson"* - è ancora l'investigatore tardo-vittoriano che parla - *"una volta eliminato l'impossibile, quello che resta, anche se incredibile, deve essere la verità: dato che è impossibile che la dichiarazione di un puntatore a Seconda sia la stessa cosa che dichiarare un puntatore a Prima, si deve concludere logicamente che a decidere quale versione di scrivi\_tutto vada eseguita, quando non è indicato il risolutore di ambito, sia la classe del puntatore che sta a destra dell'operatore di assegnamento."*   
  
Questa volta Holmes ha quasi indovinato; per l'esattezza le cose stanno così: quando almeno un metodo di una classe, non importa se pubblico o no, è qualificato con la parola *virtual*, quella classe, con tutte le sue eventuali eredi, è denominata classe/gerarchia polimorfa.  
  
Quando, in una gerarchia polimorfa, un puntatore a una classe antenata viene inizializzato con l'indirizzo di un oggetto di (ovvero con un puntatore a un oggetto di) una classe erede, SE il puntatore antenato così inizializzato viene usato per richiedere che sia eseguito il metodo *virtual*, SENZA CHE DI ESSO SIA ESPLICITAMENTE INDICATO IL RISOLUTORE DI AMBITO, il compilatore, mentre sta compilando, segnala al *linker* che tale invocazione non deve essere risolta al momento della compilazione, ma SOLO durante l'esecuzione del programma. In quella fase, NON PRIMA, la richiesta di esecuzione sarà risolta mandando in esecuzione la cosiddetta sovrascrittura finale (*final overrider*) del metodo, che sia tale al momento in cui l'esecuzione stessa debba essere avviata.  
  
Siccome le parole sono pietre, lasciatevi lapidare dal precedente capoverso rileggendolo tante volte quante ne occorrono a realizzare un censimento preciso di ogni parola di cui non avete compreso a fondo il senso, dopo di che proseguite la lettura.  
  
Innanzitutto va sottolineato che la qualifica *virtual* è essa stessa ereditata, il che significa che, nel nostro esempio, tutte le funzioni *scrivi\_tutto( )*, a qualunque classe della gerarchia appartengano, sono qualificate *virtual*, anche se non viene scritto esplicitamente; peraltro la ripetizione dell'attributo *virtual* nelle dichiarazioni/definizioni di *scrivi\_tutto( )* poste nelle classi eredi non è un errore, anzi potrebbe contribuire alla chiarezza e alla comprensibilità del programma (il compilatore, comunque, se ne infischia). Questo spiega come mai si sia potuto usare il termine metodo *virtual* per identificare, nel presente esempio, una qualsiasi delle *scrivi\_tutto( )*, quando INVOCATE SENZA RISOLUTORE DI AMBITO.   
  
In secondo luogo mette conto puntualizzare che la virtualizzazione coinvolge SOLAMENTE quell'*overload* di *scrivi\_tutto( )* che per primo è stato infettato con la parola *virtual*: se nella classe *Prima* fosse contenuto un secondo *overload* di *scrivi\_tutto( )* (ad esempio *scrivi\_tutto(int k)*) NON qualificato *virtual*, questo secondo *overload* sarebbe un metodo come tutti gli altri che si sono sempre visti, e sarebbe escluso dal meccanismo che si sta discutendo. Lo stesso dicasi di ogni eventuale altro *overload* che fosse presente nelle classi eredi.  
  
In terzo luogo, come l'esempio mostra con chiarezza, la presenza esplicita del risolutore di ambito AZZERA il meccanismo di cui si sta parlando: quando nel codice si richiede l'esecuzione di *Prima::scrivi\_tutto( )*, è SEMPRE **QUELLA** funzione a essere eseguita, qualunque sia il tipo del puntatore usato e/o il tipo del puntatore inizializzante.  
  
Il polimorfismo della gerarchia inizia sempre dalla *più antica* classe con metodi *virtual*; nel nostro esempio, se aggiungessimo una classe ancestrale *Zeresima*, da cui *Prima* ereditasse, e nella classe *Zeresima* fosse inserito un metodo *scrivi\_tutto( )* NON qualificato *virtual*, il polimorfismo continuerebbe a iniziare da *Prima* e un eventuale puntatore a *Zeresima* continuerebbe a poter invocare SOLO IL SUO *scrivi\_tutto( )*, quand'anche venisse inizializzato con un puntatore a *Terza*.  
  
Il polimorfismo si esplica SOLO ATTRAVERSO I PUNTATORI O I RIFERIMENTI, **MAI TRAMITE GLI OGGETTI**: un OGGETTO della classe *Prima*, al contrario di un puntatore o un riferimento alla classe *Prima*, NON POTRÀ **MAI** ESSERE USATO PER ESEGUIRE UN METODO DELLA CLASSE *Terza*, che sia o no virtuale, semplicemente perché NON CE L'HA; piuttosto potrà accadere il contrario, ossia che un oggetto della classe *Terza* esegua un metodo della classe *Prima*, altrettanto semplicemente perché lo ha ereditato.  
  
Tutto ciò consolidato e impresso **a caratteri di fuoco** nel vostro cervello, veniamo a descrivere dettagliatamente come funziona la baracca, perché si tratta di un'altra serissima motivazione per poter dire GRAN LINGUAGGIO.  
  
Allorché, in un dato livello di una gerarchia ereditaria di classi, punge vaghezza al programmatore inserire in una certa classe ANCHE UNA SOLA funzione membro *virtual* (il che significa che lo stesso programmatore ha facoltà di inserirne quante ne voglia) quella classe, e TUTTE le sue eredi, come si è detto, diventano polimorfe.  
  
A quel punto, TUTTE LE EREDI, che hanno comunque facoltà, NON OBBLIGO, di ridefinire nel proprio ambito, quand'anche lo ereditassero, una diversa versione dello stesso *overload* di QUEL metodo, come si è visto fin dall'inizio del presente canto, SE LO RIDEFINISCONO quella versione ridefinita NON È SEMPLICEMENTE UN'ALTRA FUNZIONE MEMBRO, come avviene per le funzioni membro normali e come si è visto accadere all'inizio di questo canto prima che si usasse *virtual*, MA È LA SOVRASCRITTURA FINALE (appunto il *"final overrider"*), relativa alla classe in cui si trova, del metodo *virtual*, al punto che il compilatore accetta persino che nella dichiarazione/definizione del *final overrider* sia esplicitamente inserita la parola *override*, per sottolineare al lettore del codice (il compilatore, come si è visto, ne fa serenamente a meno) il ruolo svolto dal metodo così marcato. In altre parole, dalla classe *Seconda* in poi, il metodo *scrivi\_tutto* avrebbe potuto avere anche la seguente dichiarazione/definizione:  
  
*void scrivi\_tutto( ) override  
{/\* tutto quanto sta nella funzione \*/}*   
  
o anche, ripetendo ed esemplificando quanto è già stato scritto,   
  
*virtual void scrivi\_tutto( ) override  
{/\* tutto quanto sta nella funzione \*/}*   
  
La parola *override*, proprio per il suo essere facoltativa, NON È UNA PAROLA DI VOCABOLARIO, in senso stretto, tanto che potrebbe tranquillamente essere usata anche come nome per una propria variabile; il suo inserimento nella posizione indicata va inteso come un ausilio per il programmatore, dato che il compilatore si premurerà di segnalare ERRORE se viene usata a sproposito nella dichiarazione/definizione di un metodo che NON POTREBBE ESSERE il *final overrider* di checchessia, magari semplicemente perché si sta definendo un diverso *overload*, o comunque perché non c'è, nella gerarchia, ALCUN METODO *virtual* completamente omonimo...o perché...fra un po' ve lo dico.  
  
Si è detto che le eredi di una classe polimorfa NON HANNO OBBLIGO di definire un *final overrider* nel proprio ambito; quando non lo fanno, PER LORO il *final overrider* è quello definito dall'antenata che è loro più vicina in linea ascendente DIRETTA, NON in eventuali linee ereditarie collaterali. Al limite estremo, SE NESSUN'EREDE RIDEFINISSE IL METODO *virtual* PRIMIGENIO, quest'ultimo finirebbe per essere il *final overrider* per TUTTE LE EREDI, ma in questo caso, con ogni evidenza, la parola *virtual* stessa sarebbe depauperata di ogni significato.  
  
Si è anche detto che il metodo qualificato per primo come *virtual* (nell'esempio corrente *Prima::scrivi\_tutto( )*) non ha bisogno di essere pubblico, anzi potrebbe addirittura essere privato. In questo caso, come ben si sa, nemmeno può essere ereditato, ma nondimeno la definizione di un *final overrider* continua a poter essere realizzata nella sua piena funzionalità. Occorre essere molto chiari su questo punto: NON si sta dicendo che, se si trasla la prima definizione di *scrivi\_tutto( )* nella zona privata della classe *Prima*, il programma dell'esempio si può ancora compilare; basti pensare, se non altro, che perderebbero legittimità TUTTE le richieste di esecuzione di *Prima::scrivi\_tutto( )* sparse in *main*. Ma la perderebbero anche quelle del *final overrider* delle classi eredi. E allora? Che significa la frase *"la definizione di un* ***final overrider*** *continua a poter essere realizzata"* quando il metodo *virtual* primordiale fosse privato?  
  
Significa esattamente quello che dice, ma occorre comunque un'interfaccia pubblica per eseguirlo, in ossequio alle regole fondamentali sull'accesso ai membri di una classe. Un paio di ulteriori esempi fatti apposta aiuteranno a digerire i concetti.  
  
**Esempio 1:** (sul *final overrider* di un metodo PRIVATO)

# include <iostream>  
using namespace std;  
  
class Prima  
{// omissis  
virtual void scrivi\_tutto(  )  
{cout  
<< "io sono il metodo virtual PRIVATO \n"  
<< "prima = " << prima  
<< '\n';}  
protected:  
int prima;  
public:  
void interfaccia\_pubblica(  )  
{scrivi\_tutto(  );}  
Prima(int i) {prima = i;}  
Prima(  ) {prima = 33;}  
};  
  
class Seconda : public Prima  
{// omissis  
protected:  
int seconda;  
public:  
Seconda(int i) : Prima(i) {seconda = i\*2;}  
Seconda(  ) {seconda = 66;}  
virtual void scrivi\_tutto(  ) override  
{cout  
<< "io sono il \"final overrider\" di un metodo PRIVATO virtual\n"  
<< "prima = " << prima  
<< "; seconda = " << seconda  
<< '\n';}  
};  
  
int main(  )  
{  
Prima \*p = new Prima(1);  
Seconda ss(2);  
Prima &s = ss;  
p -> interfaccia\_pubblica(  ),  
cout << "\_\_\_\_\_\_\_\n\n",  
s . interfaccia\_pubblica(  );  
cout << "\_\_\_\_\_\_\_\n\n",  
ss . interfaccia\_pubblica(  );  
cout << "...che SCOPERTA!\n";  
}

Questo codice, una volta eseguito, provvede da solo a commentarsi e anche a mostrare, come era stato anticipato, che il meccanismo della virtualizzazione si attua anche tramite i riferimenti, non solo tramite i puntatori, come appare chiaro dall'utilizzo del riferimento *&s*, di tipo *Prima*, all'oggetto *ss*, di tipo *Seconda*.  
Che poi quest'ultimo, quando esegue il metodo pubblico ereditato *interfaccia\_pubblica*, cagioni, dall'interno di tale metodo, l'esecuzione della PROPRIA *scrivi\_tutto* non è di certo una sorpresa, come *main* stessa, con una puntina di sottile sarcasmo, si affretta a evidenziare.  
  
Un altro fatto che questo esempio sottopone alla vostra meditazione è che il meccanismo della virtualizzazione dei metodi membri consente a un metodo antenato (*interfaccia\_pubblica*) l'accesso a variabili delle classi eredi (la variabile membro *seconda*), della cui esistenza la classe madre (*Prima*), cui il metodo appartiene, è, e DEVE essere, del tutto ignara.  
  
**Esempio 2:** una gerarchia con rami collaterali

# include <iostream>  
using namespace std;  
  
struct Ancestrale  
{// omissis  
virtual void mi\_presento(  )  
{cout << "io sono il metodo virtual Ancestrale \n";}  
Ancestrale(  ) {  }};  
  
struct Antenata\_1 : Ancestrale  
{// omissis  
Antenata\_1(  ) {  }  
void mi\_presento(  )  
{cout  
<< "io sono il \"final overrider\" Antenata\_1\n";}};  
  
struct Antenata\_2 : Ancestrale  
{// omissis  
Antenata\_2(  ) {  }  
virtual void mi\_presento(  )  
{cout  
<< "io sono il \"final overrider\" Antenata\_2\n";}};  
  
struct Antenata\_3 : Ancestrale  
{// omissis  
Antenata\_3(  ) {  }  
// costei non definisce un "final overrider"  
};  
  
struct Figlia\_1 : Antenata\_1  
{// omissis  
Figlia\_1(  ) {  }  
void mi\_presento(  ) final  
{cout << "io sono il \"final overrider\" Figlia\_1\n"  
"e sono quello definitivo del ramo ereditario \_1\n";}  
};  
  
struct Nipote\_1 : Figlia\_1  
{// omissis  
Nipote\_1(  ) {  }  
/\*  
void mi\_presento(  ) override  
{cout << "io sarei il \"final overrider\" Nipote\_1\n"  
"ma SONO un ERRORE\n";}\*/  
};  
  
struct Figlia\_2 : Antenata\_2  
{// omissis  
Figlia\_2(  ) {  }  
// io NON definisco un "final overrider"  
};  
  
struct Nipote\_2 : Figlia\_2  
{// omissis  
Nipote\_2(  ) {  }  
void mi\_presento(  ) override  
{cout << "io sono il \"final overrider\" Nipote\_2\n";}  
};  
  
struct Figlia\_3 : Antenata\_3  
{// omissis  
Figlia\_3(  ) {  }  
// io NON definisco un "final overrider"  
};  
  
struct Nipote\_3 : Figlia\_3  
{// omissis  
Nipote\_3(  ) {  }  
virtual void mi\_presento(  ) override  
{cout << "io sono il \"final overrider\" Nipote\_3\n";}  
};  
  
int main(  )  
{  
Ancestrale \*p;  
enum classi  
{ancestrale,  
antenata\_1, antenata\_2, antenata\_3,  
figlia\_1, figlia\_2, figlia\_3,  
nipote\_1, nipote\_2, nipote\_3};  
int i;  
clog << "Scegli che cosa istanziare:\n"  
"0: oggetto ancestrale\n"  
"1: oggetto antenata\_1\n"  
"2: oggetto antenata\_2\n"  
"3: oggetto antenata\_3\n"  
"4: oggetto figlia\_1\n"  
"5: oggetto figlia\_2\n"  
"6: oggetto figlia\_3\n"  
"7: oggetto nipote\_1\n"  
"8: oggetto nipote\_2\n"  
"9: oggetto nipote\_3\n", cin >> i;  
switch(static\_cast<classi>(i))  
{default:  
cout << "scelta errata: ANALFABETA!\n"; return 222;  
case ancestrale:  
p = new Ancestrale; break;  
case antenata\_1:  
p = new Antenata\_1; break;  
case antenata\_2:  
p = new Antenata\_2; break;  
case antenata\_3:  
p = new Antenata\_3; break;  
case figlia\_1:  
p = new Figlia\_1; break;  
case figlia\_2:  
p = new Figlia\_2; break;  
case figlia\_3:  
p = new Figlia\_3; break;  
case nipote\_1:  
p = new Nipote\_1; break;  
case nipote\_2:  
p = new Nipote\_2; break;  
case nipote\_3:  
p = new Nipote\_3;}  
p -> mi\_presento(  );  
}

Avendovi condotto, passo passo, fin qui, quest'ultimo è l'esempio che dissiperà definitivamente ogni vostro dubbio e vi schiuderà per sempre l'uscio della piena conoscenza della virtualizzazione e di OGNI PAROLA che è stata scritta in questo canto, facendovene profondamente comprendere il significato e l'utilità.  
Se la Natura vi ha equipaggiato dell'ottava virtù del buon programmatore in misura sufficiente, o, in difetto di tale lascito naturale, vi industrierete a coltivarla, con fatica e col contributo della vostra volontà, non vi sarà difficile immaginare la vastità e la varietà delle possibili applicazioni.  
  
...se poi i metodi virtuali fossero più di uno solo, e non così racchi come *mi\_presento*...  
  
...e se la gerarchia fosse costituita di classi templatizzate...   
  
...e se addirittura fossero templatizzate variadiche, con non si sa quante specializzazioni, totali o parziali...   
  
**IMMAGINATE...POTETE!** (G. Clooney)  
  
**ESEGUITE DUNQUE CON REVERENZA** questo programma **NUMEROSE VOLTE**, operando scelte diverse in ogni esecuzione, SENZA MAI RICOMPILARE tra un'esecuzione e l'altra...capirete TUTTO.  
  
Per chi, nondimeno, avesse difficoltà di piena comprensione, eccovi il solito elenco puntato che vi sottolineerà che cosa avreste dovuto capire anche senza:

1. Nel programma è stata usata una gerarchia di *struct* SOLO in omaggio alla pigrizia, che ha suggerito come evitare di dover esplicitare etichette *public*. Tutto sarebbe stato equivalente se la gerarchia fosse stata realizzata con la parola *class* o perfino mescolando le due parole di vocabolario, AL SOLO PREZZO di giocare correttamente con le etichette di accesso e di eredità e di introdurre, eventualmente, opportune interfacce pubbliche, come si era già visto nel precedente esempio.
2. Per come è scritto il programma, e per il poco o nulla che fa, TUTTI i costruttori di *default*, definiti esplicitamente in ciascuna classe, avrebbero tranquillamente potuto essere omessi. Non è cattiva abitudine definirli comunque, anche quando nemmeno servirebbero.
3. L'espressione che controlla *switch* sarebbe potuta essere semplicemente *i*, dato che il compilatore è capace da sé di convertire una variabile dichiarata di tipo *int* in una enumerazione come *classi*, conversione tacitamente necessaria per eseguire il confronto con le costanti previste nei diversi *case*; dopo tutto, il tipo sottostante *classi* è *int*, per l'appunto. Tuttavia l'esplicita coerenza è sempre commendevole, e poi non fa male prendere dimestichezza con gli operatori di *casting* del linguaggio: lo stesso *casting* si sarebbe potuto ottenere ANCHE con l'operatore unario classico, ossia scrivendo *switch((classi)i)*
4. Nel programma esiste UNA SOLA richiesta di esecuzione del metodo *mi\_presento(  )*, situata nell'ultima riga di *main*, e tale richiesta si compie tramite un puntatore alla classe *Ancestrale*...Quando il compilatore svolge il proprio ufficio e giunge a tradurre in codice binario QUELLA riga...NON HA ALCUN VALIDO CRITERIO PER CONOSCERE COME SIA STATO INIZIALIZZATO IL PUNTATORE *p*. Ecco che cosa significa dire che la risoluzione di quell'invocazione di funzione non può avvenire nel momento della compilazione, ma SOLO nel momento dell'esecuzione, allorchè qualche scimmia catarrina digita qualcosa sulla tastiera e, sperabilmente, il puntatore viene inizializzato in qualche modo, districando il flusso d'esecuzione attraverso il labirinto *switch*. Ed ecco, in definitiva, perché esiste la parola *virtual* e tutto l'*ambaradan* a essa collegato.
5. Nella scrittura del codice delle classi ci si è concessa un'ampia variabilità nell'esplicitazione od omissione delle parole *virtual* e *override* NELLE CLASSI EREDI per mettere ancora in evidenza il loro essere opzionali. In più è stata usata, nella classe *Figlia\_1*, la parola *final* che, alla stessa stregua di *override* NON È DA CONSIDERARE UNA PAROLA DI VOCABOLARIO, ma ha l'effetto di **IMPEDIRE**, pena segnalazione di errore, ulteriori definizioni di *final overriders* lungo la stessa catena ereditaria, della qual cosa opportunamente prende atto la classe *Nipote\_1*, autocommentandosi la sua abortita pretesa.  
     
   In ossequio alla variabilità evocata in questo stesso punto, la definizione del *final overrider* in *Figlia\_1*, mantenendo ferma la sua apocalitticità, avrebbe potuto indifferentemente assumere una qualsiasi delle seguenti forme equivalenti:
   * *virtual void mi\_presento(  ) final {/\*omissis\*/}*
   * *virtual void mi\_presento(  ) final override {/\*omissis\*/}*
   * *virtual void mi\_presento(  ) override final {/\*omissis\*/}*
   * ...etcetera
6. Le tre diverse linee ereditarie, tutte prosapia della classe *Ancestrale*, hanno comportamenti diversi giustappunto in accordo alle diverse ridefinizioni dei *final overriders* lungo ciascun *phylum*. Ecco che cosa significa avere scritto, in questo canto, che il *final overrider* di ciascuna classe in una gerarchia polimorfa ***è quello definito dall'antenata che è più vicina in linea ascendente DIRETTA, NON in eventuali linee ereditarie collaterali*** (autocitazione).
7. ***"NON in eventuali linee ereditarie collaterali"*** (ri-autocitazione dal punto precedente) significa, in particolare, che **NON È NEPPURE AMMISSIBILE** una dichiarazione inizializzata come questa:  
   *Antenata\_1 \*A = new Antenata\_2;*  
   (che produce ERRORE), sicché la ricerca di *final overriders* in rami collaterali è occlusa alla fonte.

Questo canto si chiude con una raccomandazione: **TEMETE** l'uso dell'operatore *delete* su un puntatore a una classe come *Ancestrale*, fondamento di una gerarchia polimorfa, almeno fino a quando non ne sarà stata compiutamente studiata e descritta l'operazione. Ecco un altro sospeso da tenere a mente: qualora mi sfuggisse di riprenderne qualcuno, SEGNALATE la cosa con sollecitudine.

**Canto trentaduesimo: dopo le virtù del programmatore...le virtualità del linguaggio; parte III: i metodi virtuali puri e le classi astratte**

*Filosofia - mi disse - a chi la 'ntende,  
nota, non pure in una sola parte,  
come natura lo suo corso prende  
  
dal divino 'ntelletto e da sua arte;  
e se tu ben la tua Fisica note,  
tu troverai, non dopo molte carte,  
  
che l'arte vostra quella, quanto pote,  
segue, come 'l maestro fa 'l discente;  
sì che vostr'arte a Dio quasi è nepote.  
  
Da queste due, se tu ti rechi a mente  
lo Genesì dal principio, convene  
prender sua vita e avanzar la gente*   
  
Questi versi sublimi, che ogni Fisico degno di questo nome dovrebbe saper mandare a memoria, sono anche appropriati a introdurre, in veste poetica di qualità impareggiabile, l'argomento di questo canto.   
  
In effetti, come *"nostr'arte a Dio quasi è nepote"*, altrettanto una gerarchia polimorfa può essere resa nepote di un ente assolutamente immateriale che, nel linguaggio C++, prende il nome, assai più umile, di *classe astratta*.  
  
*Ma seguimi oramai che 'l gir mi piace;  
ché i Pesci guizzan su per l'orizzonta,  
e 'l Carro tutto sovra 'l Coro giace*Indipendentemente da dove si trovino in cielo il Carro e i Pesci nell'ora e nel giorno in cui leggerete queste note (se le leggerete) seguitemi *"come 'l maestro fa 'l discente"*.  
  
Immateriale e astratto sono aggettivi che forniscono chiaro indizio della verità, ossia che di una *classe astratta* non è MAI POSSIBILE istanziare alcun oggetto.  
  
**DOMANDA:**  
quando o come una classe si dice astratta?  
  
**RISPOSTA:**  
una classe si dice astratta quando, fra i suoi metodi membri, ve n'è ALMENO UNO che sia qualificato *virtual* e che sia puro; si dice, di un tale metodo, che si tratta di una funzione virtuale pura (*pure virtual function*).  
  
**DOMANDA:**  
Che cos'è e come è fatta una funzione virtuale pura?  
  
**RISPOSTA:**  
Una funzione virtuale pura è una funzione virtuale che subisce una formale inizializzazione a zero, ossia la cui dichiarazione, nell'ambito della classe, assumendo il tipo restituito *void* e la lista degli argomenti vuota SOLO come campioni, ha la forma seguente:  
 *virtual void io\_sono\_una\_funzione\_virtuale\_pura(  ) = 0;*  
Una tale funzione non può essere definita contestualmente con la precedente dichiarazione perché l'apposizione di un ambito di funzione al posto del punto e virgola genererebbe un chiaro errore sintattico in concomitanza con l'inizializzazione a zero.  
  
**DOMANDA:**  
Una funzione virtuale pura può essere definita?  
  
**RISPOSTA:**  
Sì, ma senza che ve ne sia l'obbligo e comunque SOLO al di fuori dell'ambito della classe di appartenenza, adottando l'opportuna sintassi che ne usa il risolutore di ambito.  
  
**DOMANDA:**  
Si è detto che non si possono istanziare oggetti di una classe astratta; quindi, se ne viene definita una, a che cosa serve?  
**RISPOSTA:**  
Non si possono istanziare oggetti di una classe astratta, ma si possono tranquillamente dichiarare puntatori e riferimenti a una tale classe; pertanto una classe astratta può essere usata come capostipite di una gerarchia polimorfa né più né meno di come è avvenuto con la classe *Ancestrale* del canto precedente. In tal senso le classi eredi sono ad essa *"quasi nepoti"*.  
  
**DOMANDA:**  
E quali sono le differenze, i vantaggi o gli svantaggi dell'avere una gerarchia polimorfa fondata su una classe astratta piuttosto che su una classe normale, di cui si possano istanziare oggetti?  
**RISPOSTA:**  
Parlare di differenze è legittimo; parlare genericamente di vantaggi e/o svantaggi è del tutto inappropriato: sarà sempre e solo il buon programmatore a dover decidere che cosa sia vantaggioso o svantaggioso per lui/lei, in ogni particolare contesto. Naturalmente, per potere scegliere sempre il miglior partito, occorrerà che conosca le differenze fin nei minimi dettagli.  
  
**RI-DOMANDA:**  
Non menare il can per l'aia, come quella volta che ti chiesero *"Sai dirmi che ore sono?"* e rispondesti solo *"Sì"*! Riformulo la domanda: quali sono le differenze tra una gerarchia polimorfa fondata su una classe astratta e una fondata su una classe NON astratta?  
**RISPOSTA:**  
Ecco una domanda pertinente. La differenza sostanziale risiede nel seguente assioma: **Finché, tra le classi eredi, NON SE NE TROVA UNA che definisca il proprio *final overrider* relativo a una funzione virtuale pura, TUTTO il *phylum* è costituito di classi astratte**.  
Ciò è perfettamente coerente col fatto che, in assenza di ridefinizione di un *final overrider*, ne vige uno precedente; se si tratta di quello della classe capostipite, che è puro, resta astratta anche una classe erede renitente a ridefinirsi il metodo, e così via lungo l'intera gerarchia, fino a quando, per l'appunto, una certa classe erede decide di ridefinire il metodo, per ciò stesso concretizzandosi assieme a tutte le successive eredi. Si ricordi anche che quanto detto prescinde dal fatto che la funzione virtuale pura sia ereditata o no, come si è già visto nel canto 31. Per constatare la veridicità delle affermazioni contenute in questa risposta è sufficiente far diventare astratta la classe *Ancestrale*, trasformando in funzione virtuale pura il metodo *mi\_presento(  )*, e contare il numero degli errori di compilazione che ne conseguono, valutandone nel contempo le motivazioni [**FATELO!**].  
  
**DOMANDA:**  
E se è la classe capostipite stessa a definire la funzione virtuale pura? Si era detto che questo era legittimo: non basta a concretizzare tutta la gerarchia e, in ultima analisi, anche a poter istanziare oggetti della classe capostipite?  
**RISPOSTA:**  
NO. L'eventuale definizione, da parte della classe capostipite, della funzione (o delle funzioni) virtuale pura che la rende astratta NON NE MODIFICA IN ALCUN MODO LA DICHIARAZIONE, che è SEPARATA dalla definizione e resta, IN OGNI CASO, quella di una funzione virtuale pura.   
Diverso è il discorso per le classi eredi, le quali eventualmente ridefiniscono, **NON MAI solamente ridichiarano**. In sostanza l'illibatezza di una classe astratta capostipite non può mai andare perduta.  
  
**DOMANDA:**  
Ma allora a che cosa potrebbe mai servire la definizione di un metodo virtuale puro in una classe astratta capostipite? Oggetti di quella classe non ce ne sono; neppure ci possono essere oggetti di classi eredi fino a quando il metodo non viene definito; la classe che lo definisce per prima dispone, per tautologia, della propria ridefinizione, valida anche per tutte le classi eredi che le succedono e che non lo ridefiniscano a propria volta...e quindi???   
**RISPOSTA:**  
Ricordato che comunque NON È NECESSARIO che la classe capostipite definisca i propri metodi virtuali puri, avete dimenticato la possibilità di richiedere l'esecuzione di un metodo citandolo col risolutore di ambito? Se la funzione virtuale pura, eventualmente definita nella classe capostipite, passa in eredità alle classi figlie, o è comunque accessibile nella classe capostipite, la sua invocazione, se completa di risolutore di ambito, evita, se si può dire, ogni virtualizzazione. Allora la definizione della funzione virtuale pura primigenia potrebbe servire a compiere azioni utili a qualsiasi componente della dinastia, indipendentemente dal livello occupato nella successione ereditaria.   
  
**DOMANDA:**  
In definitiva, si può sapere perché è uscita fuori questa pensata delle classi astratte?  
**RISPOSTA:**  
Serve, in sostanza, e visto quanto si è sottolineato circa la differenza con una gerarchia polimorfa NON fondata su una classe astratta, a **IMPORRE** delle regole a chi si inserisce nella linea ereditaria, dato che ALMENO UN'EREDE **DEVE** assumersi la responsabilità di ridefinire il metodo virtuale puro, pena l'inutilizzabilità dell'intera gerarchia. Se il buon programmatore decide di scrivere una classe che si appoggi su una classe astratta, **SA** che gli/le tocca scriversi anche, esplicitamente, TUTTI i metodi virtuali puri che la classe astratta prevede, OBBEDENDO SCRUPOLOSAMENTE alla loro segnatura. A sua volta egli/ella ha facoltà di inserire propri metodi virtuali puri nella classe erede, rendendola per ciò stesso astratta, e imponendo quindi LE PROPRIE REGOLE a chiunque volesse utilizzarla come antenata.  
  
Alla fine di questa breve catechesi, si spera che le classi astratte non conservino più alcun mistero per nessuno di voi.

**Canto trentatreesimo: i tortuosi percorsi del *casting***

**Premessa**  
  
Alle operazioni di *casting* si dovrebbe poter ricorrere il meno possibile, e sempre e comunque come *extrema ratio*, in quanto dovrebbe sempre poter esistere una maniera alternativa di scrivere il codice, tale da poterne fare tranquillamente a meno.  
Essenzialmente le operazioni di *casting* risultano utili, o quasi necessarie, SOLO quando si tratta di dover trasmettere a una funzione già compilata, e della cui sorgente di codice non si abbia quindi contezza, un certo parametro e non ce ne sia disponibile uno di tipo conforme all'argomento atteso, ma solo qualche variabile di tipo compatibile.  
  
Anche l'affermazione di quest'ultimo capoverso va tuttavia presa con una certa elasticità (non per nulla si è scritto *"quasi necessarie"*) dato che, sovente, il compilatore riesce a effettuare le dovute conversioni ANCHE in assenza di qualsiasi *casting* esplicito. Ad esempio il seguente codice scrive correttamente sullo *standard output* le prime sei cifre significative del numero di Nepero, nonostante la sua **PESSIMA calligrafia**, che richiede al compilatore ben DUE operazioni implicite di *casting* (riuscite a vederle?):   
  
*# include <iostream>  
# include <cmath>  
using namespace std;  
  
void funza(int x)  
{  
cout << exp(x) << endl;  
}  
  
int main( )  
{  
double n = 1.0;  
funza(n);  
}*La calligrafia del precedente programma è **PESSIMA** perchè, se vi illudete di poter ottenere il valore dell'esponenziale di 1.999999 semplicemente sostituendo quest'ultimo numero a 1.0 [PROVATECI] riceverete una sgradita sorpresa...  
  
Le conversioni implicite sottostanno anch'esse a regole ben determinate che, spesso, vengono passate sotto silenzio data la loro quasi totale ovvietà; qui tuttavia saranno passate in rassegna in modo da poter eliminare anche il quasi, distinguendole in:

1. **trasformazioni di categoria**; sono quelle in cui un *lvalue* viene tacitamente trasformato in *rvalue* o viceversa (questa voce viene scritta per completezza: di *lvalues* e *rvalues* si parlerà più diffusamente nel futuro canto 36). Avvengono nei seguenti contesti:
   * *lvalue* utilizzato al posto di un numero (*prvalue*) come in  
     *const int i = 20; // i è un lvalue  
     double a[i]; // i usato dove dovrebbe essere un prvalue*
   * puntatore (*lvalue*) usato al posto di un array (*rvalue*) come in  
     *void funza(int \*pt) { } // funzione che riceve un lvalue  
     int main( ) {int a[2]; funza(a);} // e a cui viene trasmesso un rvalue  
     // (e viceversa)*
   * nome di una funzione usato al posto di un puntatore a funzione di ugual segnatura come in:  
     *void funza(double (\*a)(double), double x) {cout << a(x) << '\n';}  
     int main( ) {funza(sin, 1.57);}*in cui, tra l'altro, si rivede un *prvalue*, la costante 1.57, tacitamente trasformata in un *lvalue*, la variabile *x* in *funza*
2. **promozioni numeriche**; sono tutte quelle (forse le più comuni) in cui una variabile appartenente a un tipo numerico viene trasformata implicitamente in una di tipo più esteso, e senza cambiamento di valore, come in:  
     
   *char a = 'a';  
   int b = 1 + a;  
   float f = 1.f;  
   double d = f + 1.0;  
   // et cetera similia*
3. **conversioni numeriche**; sono quelle in cui la trasformazione di una variabile di tipo numerico avviene in verso opposto rispetto al punto precedente, con perdita di precisione ed eventuale cambiamento di valore, come in:  
     
   *double d = 1.24;  
   float f = d; // il valore 1.24 si conserva, con solo perdita di precisione  
   int i = d + 1; // perdita della parte decimale di 2.24  
   int k = 12345678;   
   unsigned short u = k; // u diventa 24910 (!)*
4. **conversione di puntatori**, compresi i puntatori a membro, da distinguere in:
   * conversione del puntatore nullo *nullptr*, o costante equivalente, al puntatore nullo di qualsiasi tipo, compreso il puntatore a membro nullo del tipo del membro puntato, come in:  
       
     *double \*a = nullptr;  
     struct Ciccio {int i;};  
     int Ciccio::\*pt = &Ciccio::i;  
     pt = nullptr;*
   * conversioni di puntatori a oggetti di una classe erede, o di puntatori a membro di una classe erede, in puntatori a una classe antenata o a membri di una classe antenata che siano non meno qualificati rispetto alle qualifiche *const* e/o *volatile*, come in:  
       
     *struct Antenata {int i = 4;};  
     struct Erede : Antenata {};  
     int main( )  
     {Antenata \* antenata = new Erede; // OK, identicamente qualificato  
     const Antenata \* antenata\_bis = new Erede; // OK, più qualificato  
     // Antenata \* antenata\_ter = new const Erede;  
     // ERRORE, meno qualificato  
     int Erede::\*pt = &Antenata::i; // puntatore a membro di Erede  
     // inizializzato con un puntatore a membro di Antenata  
     Erede e;  
     cout << e .\* pt << ' ' << antenata->i + 1 << '\n';   
     }***Osservazione:** per la conversione di un puntatore a membro, la classe *Antenata* dell'esempio NON DEVE essere un'antenata virtuale.
   * conversioni di puntatori a variabili/oggetti (**NON** di puntatori a membro) in puntatori a *void*, sempre possibili con la clausola della non minore qualifica *const* e/o *volatile*, come in:  
       
     *struct Pappo {int i = 4;};  
     int main( )  
     {void \*pt = new Pappo; // OK: identicamente qualificato  
     // per poter usare pt, ad esempio:  
     char \*pq = reinterpret\_cast<char\*>(pt); // cfr. appresso  
     for(int i=0; i < sizeof(Pappo); ++i) cout << (int)pq[i] << '\n';  
     // pt = &Pappo::i; // ERRORE: puntatore a membro  
     // pt = new const Pappo; // ERRORE: minor qualificazione  
     }*
5. **conversioni di qualificazione**; in parte già accennate al punto precedente consistono non tanto nel cambiamento di tipo quanto, appunto, nella diversa qualificazione tramite le parole di vocabolario *const* e/o *volatile*. Le regole auree sono che non è ammissibile che la qualificazione divenga minore a causa della presente conversione, e che *const* e *volatile* possono cumularsi ma NON SOSTITUIRSI A VICENDA. La relazione ordinale è dunque la seguente:  
     
   nessuna qualifica < *const* OPPURE *volatile* < *const* ASSIEME A *volatile*   
     
   In pratica:  
     
   *int k = 1; const int i = k; // OK. più qualificato  
   // int \*j = &i; // ERRORE: meno qualificato  
   // volatile int \*n = &i; // ERRORE const non diventa volatile  
   const volatile int\*m = &i; // OK. più qualificato*Qualche parola in più va spesa, a proposito del presente punto di questo elenco, per i puntatori multidimensionali: quelli che hanno più di un solo asterisco nella loro dichiarazione, per capirsi.  
   Se si hanno due di tali puntatori, la conversione di qualificazione dell'uno nell'altro è legittima se:
   * ovviamente il tipo finale degli oggetti puntati è IDENTICO, altrimenti si tratta di un altro genere di conversione, cfr. appresso;
   * il numero di livelli di puntamento, ossia il numero di asterischi, è, altrettanto ovviamente, IDENTICO, altrimenti si tratta sempre di un errore. Sia *N* questo numero comune di livelli, con *N > 1* come anticipato;
   * si numerino i livelli, come d'abitudine in C++, da 0 (zero) a *N* partendo dal livello di puntamento più interno, a destra dell'asterisco più a destra, e fino al livello più esterno, a sinistra dell'asterisco più a sinistra ...
   * A ogni livello di puntamento *k > 0*, ossia escluso il livello zero che obbedisce alle regole già enunciate, la qualifica dei due puntatori è IDENTICA;
   * In alternativa, se a un certo livello di puntamento *s* si riscontra una DIVERSA QUALIFICA, allora OCCORRE che il puntatore destinatario della conversione ABBIA TUTTI I LIVELLI da 1 a *s* compreso, ossia escluso ancora il livello zero, qualificati *const*.

In pratica:  
  
*int \*l[  ]{new int[2], new int[2]};  
int \*\*ii = l;  
// const int \*\*ll = ii; // ERRORE  
int \* const \* kk = ii; // OK  
int \*\*\*iii = new int\*\*;  
iii[0] = ii; // OK. identicamente qualificato  
int\*\*\*const kkk\_0 = iii; // OK. più qualificato a livello 0  
int\*\*const\* kkk\_1 = iii; // OK. più qualificato SOLO a livello 1  
//int \*const\*\* kkk\_2 = iii; // ERRORE. più qualificato SOLO a livello 2  
int \*const\*const\* kkk\_2\_ = iii; // OK. più qualificato a livelli 1 E 2  
//const int\*\*\*kkk\_3 = iii; // ERRORE. più qualificato SOLO a livello 3  
// etcetera....*

1. **conversione booleana**; è quella adombrata dal motto *"TUTTO È VERO QUEL CHE NON È ZERO"*, ma a questo livello di conoscenza bisogna aggiungere qualche puntualizzazione che concerne il problema del cosiddetto *"safe bool idiom"*, che nasce dall'ovvia considerazione che, quando si passa a discutere di classi definite dal programmatore, non appare immediatamente ovvio che cosa significhi precisamente essere o non essere zero.  
   Per persuadersi della ragionevolezza di questo dubbio amletico basta osservare che, se per una certa classe *Pappo*, fosse definito un *operator bool( )* in maniera del tutto *naïve*, questo stesso *operator* consentirebbe anche operazioni di confronto, o altre, probabilmente prive di significato, o almeno con un significato NON contemplato, o NON previsto, dalla volontà del programmatore; si consideri il seguente esempio, volutamente provocatorio, anche se, tutto sommato, le perplessità espresse dal codice non sarebbero poi così preoccupanti, nel caso specifico:  
     
   *# include <iostream>  
   using namespace std;  
     
   class Pappo  
   {int i;  
   public:  
   operator bool(  ) {return i != 0;} // questo consente di tutto...  
   Pappo(int k = 1) : i(k) {  }  
   };  
     
   int main(  )  
   {  
   Pappo pappo;  
   if(pappo) cout << "Costruito un oggetto Pappo\n";  
   Pappo pappino(0);  
   if(!pappino) cout << "non costruito (\?\?\?) un oggetto Pappo\n";  
   cout << (pappo << 2) << '\n';  
   cout << "che diamine significa pappo << 2 (\?\?\?)\n";  
   int z = pappino, w = pappo; // inizializzare degli int con dei Pappo ???  
   if(pappo > pappino) cout << "che significa pappo > pappino (\?\?\?)\n";  
   }*  
     
   Il programma presentato **si compila** con la più grande serenità da parte del compilatore e, beninteso, se questa fosse la volontà del programmatore, non ci sarebbe nulla da obiettare sulla base delle regole del linguaggio.  
   Qualche incongruenza è tuttavia palese, specialmente se si nota che tanto *pappo* quanto *pappino* sono entrambi costruiti con pieno successo e pertanto l'uso primario che vien fatto dell'*operator bool*, ossia come bersaglio dei due *if*, appare alquanto scricchiolante.  
   Per questa ragione, ad esempio nelle classi di *input/output*, si è preferito definire piuttosto l'*operator!* e l'*operator void\** da usare rispettivamente in espressioni come *if(!oggetto\_stream)* o *if(oggetto\_stream)* per porre sotto controllo lo stato di salute di un oggetto gestore di *input* o *output*.  
   Ma, come diceva don Ferrante, *"per evitar questa Scilla ...* omissis *... dànno in Cariddi"* perché, con tale scelta, il compilatore fa passare quasi in cavalleria, come accettabile, una linea di codice che recitasse  
     
   *delete cout; // ???????????*provate e vedrete: otterrete solo un avvertimento, ma l'eseguibile sarà ugualmente prodotto ... salvo poi far abortire il programma all'atto dell'esecuzione.  
   È pur vero che chi scrivesse nel proprio codice un obbrobrio del calibro di *delete cout;* meriterebbe, oltre all'interruzione del programma, la **gogna perpetua**, ma rimane un'inquietudine di fondo: possibile che un GRAN LINGUAGGIO non contempli una *conversione booleana* che non sia prona a questo genere di inconvenienti?  
   Il *safe bool idiom* è appunto la risposta che scioglie questa inquietudine e che permette una *conversione booleana* scevra da controindicazioni, nel senso che cagiona ERRORI IN COMPILAZIONE quando venisse usata in maniera impertinente (sarebbe NON pertinente, ma rende meglio l'idea).  
   Data l'estrema rarità dei casi in cui il ricorso al *safe bool idiom* è auspicabile non starò a farvela lunga sulla sua implementazione, limitandomi a dire che esiste e ad avere spiegato, spero, il motivo per cui esiste. Chi fosse interessato lo chieda a voce.

**Fine della premessa**  
  
Le operazioni di *casting*, già incontrate qua e là lungo questo percorso, trasformano il tipo di una sottoespressione, provvisoriamente e limitatamente all'espressione in cui sono usate, **in un altro tipo**. Escludendo i casi in cui tali trasformazioni sono compiute gratuitamente dal compilatore come in tutti i discorsi fatti nella premessa, denominate appunto *casting* implicito, e che RESTANO a rischio e pericolo del programmatore, come ci si dovrebbe esser persuasi, occorre ora discutere le situazioni in cui il programmatore le opera esplicitamente, e quindi volontariamente.  
Prima della nascita del C++, il C ANSI aveva un solo operatore di *casting*, la cui sintassi era (ed è)   
  
*(tipo\_risultante)espressione*  
  
Ad esempio, nella dichiarazione immediatamente inizializzata  
*char A = (char)65;*   
la variabile *A* assume il valore di una lettera A maiuscola (cosa che peraltro avverrebbe anche con un *casting* implicito) perché è tale il valore della costante intera 65 trasformata, tramite *casting*, nel tipo *char*.  
  
Nel C ANSI, giova ricordarlo, esistevano SOLO i tipi che oggi siamo abituati a denominare nativi: neppure le *struct*, le *enum* o le *union*, che pure già esistevano, erano totalmente equiparate a tipi, tanto che potevano tranquillamente non avere neppure un nome dedicato (cosa che ANCHE in C++ PERMANE possibile).  
  
All'avvento del C++, con la sua infinità numerabile di tipi possibili, delle più varie forme e delle più diverse estensioni in memoria, e con l'aggiunta delle complicazioni apportate dai concetti di tipi eredi, tipi astratti, gerarchie polimorfe, tipi templatizzati, tipi variadici...e forse me ne sono dimenticata qualcuna... la frase *"trasformare un tipo in un altro"* appare, quanto meno, un filino azzardata, specialmente se non si specifica QUALE tipo sia da trasformare in QUALE altro.  
  
Per questa ragione il C++ ha introdotto nuovi operatori di *casting*, idonei ad affrontare e, quasi sempre, risolvere ogni possibile situazione.   
  
Tanto per cominciare, ANCHE per i tipi nativi, è stata introdotta la possibilità di operare il *casting* imitando l'invocazione di un costruttore monoparametrico, di modo che la precedente dichiarazione, con immediata inizializzazione, della variabile *A* si può scrivere nella forma pressoché equivalente  
*char A(65);*  
o anche  
*char A = char(65);*  
vale a dire con la sintassi  
  
*tipo\_risultante(espressione)*  
  
La parola *pressoché* implica, secondo logica, che l'equivalenza non è TOTALE: infatti, quando al posto di *char* ci fosse una classe definita dal programmatore, le due forme citate di *casting* si risolverebbero nell'invocazione di funzioni membro DIVERSE: la prima un metodo *operator tipo\_risultante(tipo\_espressione)* (seguita, eventualmente, dalla *operator=*; qui *tipo\_espressione* va inteso come un autocommento), la seconda un costruttore parametrico.   
  
Nella forma *tipo\_risultante(espressione)*, detta tecnicamente *functional cast* per l'aspetto simile a quello dell'invocazione di una funzione, per *tipo\_risultante* si intende **una sola parola**; questo significa, ad esempio, che se *tipo\_risultante* dovesse essere *unsigned long long int* (oppure anche *int\**) l'unico modo per utilizzare la forma del *functional cast* è farsi aiutare da *typedef* come in:   
  
*typedef unsigned long long int ulla;  
ulla UU = ulla(100000000000000);  
cout << UU << ' ' << sizeof UU << ' ' << sizeof(ulla) << '\n';*In aggiunta a quanto detto fin qui, ci sono i quattro operatori di *casting*, elencati a loro tempo assieme a TUTTI gli altri operatori del linguaggio, per cui sono date addirittura delle parole di vocabolario e che qui si rielencano:

* *const\_cast*
* *static\_cast*
* *reinterpret\_cast*
* *dynamic\_cast*

L'ordine in cui sono stati elencati rispecchia quello tacitamente applicato dal compilatore allorché gli si richieda un'operazione di *casting* per mezzo dell'antico operatore nello stile dell'ANSI C, e non sia stata approntata dal programmatore un'appropriata funzione *operator*: viene applicato il PRIMO operatore di quest'elenco che RIESCE: se nessuno riesce significa che l'operazione richiesta era troppo fantasiosa, nel qual caso il programma è verosimilmente condannato a fallire o addirittura a non essere neppure compilato.  
  
La sintassi da usare è IDENTICA per ciascuno dei quattro operatori; denominando con *uno\_qualsiasi* appunto uno qualsiasi di loro, risuona così:  
 *uno\_qualsiasi<tipo\_risultante>(espressione)*   
  
ma vanno posti vincoli abbastanza precisi, come si era anticipato, sull'accoppiata *tipo\_risultante--espressione* affinché l'operazione di *casting* vada a buon fine e la precedente espressione assuma quindi EFFETTIVAMENTE il tipo *tipo\_risultante*. Quando l'operazione RIESCE, vigono le stesse regole per quanto concerne il trattamento dell'espressione risultante come valore sinistro o destro, indipendentemente da quale sia stato l'operatore coinvolto; in questo canto è prematuro parlarne.  
  
Si passa ora a spiegare il comportamento di ciascun operatore, specificando per ciò stesso in quali contesti possa essere utilizzato.

1. L'operatore *const\_cast* RIESCE quando:
   * *espressione* è un puntatore qualunque e *tipo\_risultante* è lo stesso tipo di puntatore, eventualmente qualificato diversamente, a qualsiasi livello di puntamento, con le parole di vocabolario *const* e/o *volatile*.   
     Esempio 1:  
     *const char \* s = "questa stringa è una costante";  
     char \* p = const\_cast<char \*>(s);*Esempio 2:  
     *# include <iostream>  
     using namespace std;  
       
     int main( )  
     {  
     const char \* const \* a = new const char\*[2]{"aaa", "bbb"};  
     char \*\* b = const\_cast<char \*\*>(a);  
     cout << b[0] << '\n' << b[1] << '\n';   
     /\* // LE SEGUENTI RIGHE COMMENTATE SONO DA NON FARE   
     // QUANTUNQUE IL COMPILATORE NON LE GIUDICHI ERRONEE  
     b[0] = new char[4];  
     for(int i=0; i < 3; ++i) b[0][i] = 'a' + char(i);  
     b[0][3] = 0;\*/  
     cout << b[0] << '\n' << a[0] << '\n';  
     // a[0] = new char[2]; /\* ERRORE \*/  
     }*Esempio 3:  
     *# include <iostream>  
     using namespace std;  
       
     int main( )  
     {  
     const char \* a = "aaa";  
     char \* b = const\_cast<char \*>(a);  
     cout << a << '\n' << b << '\n';  
     b[0] = 'b'; // "COMPORTAMENTO NON DEFINITO"  
     cout << a << '\n' << b << '\n';  
     }*
   * *espressione* è un riferimento sinistro di qualunque tipo e *tipo\_risultante* è un riferimento, o puntatore, dello stesso tipo, diversamente qualificato tramite le parole *const* e/o *volatile*.  
     Esempio 1:  
     *int i = 0;  
     const int & r = i; // riferimento costante: valore sinistro  
     const\_cast<int &>(r) = 1; // i è modificato  
     // r = 2; // ERRORE senza l'uso di const\_cast*Esempio 2:  
     *const int i = 0; // stavolta qualificato const  
     int \*p = const\_cast<int \*>(&i);// COMPILABILE!  
     \*p = 1; // "COMPORTAMENTO NON DEFINITO"*
   * *espressione* è un riferimento destro di qualunque tipo e *tipo\_risultante* è un riferimento destro dello stesso tipo, diversamente qualificato tramite le parole *const* e/o *volatile*. Si darà ugualmente un esempio, nonostante lungo il percorso non si sia ancora approfondito il concetto di riferimento destro.   
       
     Esempio:  
     *# include <iostream>  
     using namespace std;  
       
     int && funza( )  
     {int i = 99;  
     return move(i);}  
       
     int main( )  
     {  
     const int && i = const\_cast<int&&>(funza( ));  
     cout << i << '\n';  
     }*
2. L'operatore *static\_cast* RIESCE quando:
   * *espressione* è una valida inizializzazione per un oggetto di tipo *tipo\_risultante*, anche tenendo conto di ogni possibile *cast implicito* o invocazione di costruttori o di operatori di conversione esplicita per la classe *tipo\_risultante*, ma senza coinvolgimento di *const* e/o *volatile*.  
       
     Esempi:  
     *float f = static\_cast<float>(1);  
     Pappo p = static\_cast<Pappo>(3.14);  
     // (purché Pappo abbia un costruttore appropriato:   
     // equivarrebbe a: Pappo p = 3.14; purché non venga usata   
     // la parola explicit, nel qual caso static\_cast continua a funzionare  
     // al contrario dell'inizializzazione  
     // tramite operatore di assegnamento)*
   * reciprocamente, ed equivalentemente, eseguendone in tal caso la trasformazione inversa, esiste un'univoca maniera di convertire implicitamente *tipo\_risultante* nel tipo cui appartiene *espressione*, e sempre prescindendo da diversità di qualificazioni *const* e/o *volatile*.  
       
     Esempio:  
     *int main( )  
     {  
     int k = static\_cast<int>(4 > 0);  
     cout << k << '\n';  
     }*
   * *espressione* è un puntatore o un riferimento a una classe *A* e *tipo\_risultante* è un puntatore o un riferimento (rispettivamente) a una classe *B* che sia erede di *A*: in tal caso viene compiuto il cosiddetto *downcast*, con significato intuitivo del vocabolo, lungo una gerarchia ereditaria. Va tenuto conto che tale *downcast* si compie senza alcun controllo durante l'esecuzione del programma: pertanto è responsabilità del programmatore che l'operazione richiesta sia plausibile, dato che il compilatore la lascia comunque passare: l'asino casca, eventualmente, all'esecuzione del codice, NON al momento della compilazione.  
       
     Esempi:  
     *class A {/\*omissis\*/};  
     class B : public A {/\*omissis\*/public: void funza( ){/\*omissis\*/}};   
       
     void funza(A \*a)  
     {  
     B \*b = static\_cast<B \*>(a);  
     b -> funza( );  
     }   
       
     int main( )  
     {  
     A \*a = new A;  
     B \*b = new B;  
     funza(a); // compilabile, MA ILLEGALE  
     a = b;  
     funza(a); // compilabile E LEGITTIMO  
     }*
   * *tipo\_risultante* è un riferimento destro qualunque ed *espressione* è un valore sinistro di uguale tipo; in tal caso *espressione* viene trasformata da valore sinistro a *xvalue*. Qui si dà un esempio senza commento, rimandando al futuro la spiegazione di che cosa sia un *xvalue*.  
       
     Esempio (prendetelo così com'è, per adesso):  
     *int main( )  
     {  
     int i = 5;  
     int k = static\_cast<int &&>(i);  
     cout << k << '\n';  
     }*
   * *tipo\_risultante* è *void*; in tal caso *espressione* viene valutata, ma il suo valore viene immediatamente scartato; evidentemnete ciò può aver luogo SOLO in un'espressione irriducibile. L'utilità di una tale operazione potrà essere meglio compresa quando si siano approfonditi i concetti di cui al punto precedente.  
       
     Esempio (prendetelo così com'è, per adesso):  
     *int funza(  )  
     {cout << "io sono funza e sono stata eseguita\n";  
     return 1;}  
       
     int main(  )  
     {  
     static\_cast<void>(funza(  ));  
     }*
   * si voglia operare ogni genere di trasformazioni coinvolgenti *array* e puntatori di ugual tipo, o assegnamenti di funzioni a puntatori a funzione di uguale segnatura, o si abbiano valori sinistri da trasformare in valori destri, sostanzialmente per le stesse ragioni dei primi due punti di questo stesso elenco.  
       
     Esempio:  
     *int main(  )  
     {  
     double a[ ] {1.5, 2.4};  
     double \*p = static\_cast<double \*>(a);  
     cout << p[0] << ' ' << p[1] << '\n';  
     }*
   * *tipo\_risultante* è un tipo numerico nativo ed *espressione* è una *enum*.  
     Esempio:  
     *# include <iostream>  
     # include <iomanip>  
     using namespace std;  
       
     int main(  )  
     {  
     enum \_Ciccio {a = 7, b, c} ciccio = a;  
     enum class \_Caccio : unsigned long long {a = 0xfffffffffffffffa, b, c};  
     float d = static\_cast<float>(ciccio);  
     unsigned long long e =   
     static\_cast<unsigned long long int>(\_Caccio::a);  
     cout << fixed << setprecision(2) << d << ' ' << e << '\n';  
     }*
   * reciprocamente, *tipo\_risultante* è una *enum* ed *espressione* è un tipo numerico nativo, o anche un'altra *enum*; in questo caso, però, se il valore risultante dalla conversione NON COINCIDE con alcuna delle costanti che costituiscono gli enumeratori della *enum* di destinazione, i risultati dell'espressione coinvolta sono IMPREDICIBILI, ma solo durante l'esecuzione: programmatore avvisato, mezzo salvato.  
       
     Esempio:  
     *int main(  )  
     {  
     enum \_Ciccio {a = 7, b, c};  
     float d = 7.5f, e = 1.0f;  
     \_Ciccio f = static\_cast<\_Ciccio>(d); // OK  
     \_Ciccio g = static\_cast<\_Ciccio>(e); // ?????????????  
     switch(f)  
     {case a:  
     cout << "sono il case a\n"; break;  
     case b:  
     cout << "sono il case b\n"; break;  
     case c:  
     cout << "sono il case c\n";}  
     switch(g)  
     {case a:  
     cout << "sono il case a?\n"; break;  
     case b:  
     cout << "sono il case b?\n"; break;  
     case c:  
     cout << "sono il case c?\n"; break;  
     default:  
     cout << "evidentemente nessuno dei tre\ninfatti g = " << g <<'\n';}  
     }*
   * *espressione* è un puntatore a un membro di una certa classe e *tipo\_risultante* è quello di un puntatore a un membro di una classe antenata e di tipo uguale; tuttavia nessun controllo viene compiuto sul fatto che l'indirizzo di destinazione esista realmente: ancora una volta programmatore avvisato, mezzo salvato.  
       
     Esempio:  
     *struct A  
     {  
     int a;  
     A(  ) : a(4) {  }  
     };  
       
     struct B : A  
     {  
     int b;  
     B(  ) : b(9) {  }  
     };  
       
     int main(  )  
     {  
     int B::\*p = &B::b;  
     int A::\*r = &A::a;  
     int A::\*q = static\_cast<int A::\*>(p);  
     B b;  
     A a;  
     cout << b.\*p << '\n'; // OK (ovviamente)  
     cout << b.\*q << '\n'; // OK (grazie, static\_cast)  
     cout << a.\*q << '\n'; // ciofeca, perché non coincide...  
     cout << a.\*r << '\n'; // ...con QUESTO!  
     }*
   * *espressione* è un puntatore a *void* e *tipo\_risultante* è un puntatore a qualunque tipo; l'applicazione involutiva di questo tipo di trasformazione ripristina certamente (il compilatore lo giura sotto pena di tortura) il valore dell'indirizzo originale.  
       
     Esempio:  
     *struct A  
     {  
     int a; A(  ) {a = 4;}  
     };  
       
     int main(  )  
     {  
     A a;  
     void \*p = static\_cast<void \*>(&a);// OK, come da punto1  
     A \*aa = static\_cast<A \*>(p); //, come da punto presente...  
     cout << aa -> a << '\n'; // ... INFATTI...  
     }*
3. L'operatore *reinterpret\_cast* RIESCE:
   * anche troppe volte, rispetto alla vostra sapienza.
   * sostanzialmente in quasi tutti i casi in cui riuscirebbe anche *static\_cast*, eccettuati quelli in cui non sono coinvolti puntatori, ovvero sono coinvolte enumerazioni o il tipo *void* (ho detto *void* NON *void \**).
   * in più ANCHE quando si tentassero conversioni, in ambo i sensi, tra qualsiasi puntatore e un tipo intero adeguato a contenerne il valore.
   * in più ANCHE quando si tentassero conversioni tra puntatori a funzioni con diversa segnatura.
   * in più ANCHE quando si tentassero conversioni tra puntatori di tipi completamente diversi, ivi compresi puntatori a classi che non c'entrino NULLA una con l'altra (il compilatore, in sostanza, pensa: *"saranno.... tuoi"*).
   * in pratica in ogni caso in cui al programmatore interessi solo poter appunto interpretare diversamente (da cui il nome dell'operatore) una determinata sequenza di bit a un certo indirizzo di memoria, senza curarsi troppo dei valori che vi siano contenuti perché magari importa solo che ci sia spazio per sovrascriverli immediatamente, in coerenza col tipo di destinazione.  
       
     Esempio utile, ma facilmente sostituibile:  
     *double x[100];  
     cin . read(reinterpret\_cast<char \*>(x), 100\*sizeof(double));*(ovviamente reindirizzando lo *standard input stream* in modo che abbia supporto in un documento binario).
4. L'operatore *dynamic\_cast*, infine, è deputato alle conversioni tra puntatori a classi appartenenti a una gerarchia ereditaria polimorfa e, diversamente da come agisce *static\_cast* quando si occupa dello stesso lavoro (vedi sopra), **EFFETTUA CONTROLLO** sulla buona riuscita della conversione, di modo che è molto più sicuro rispetto a *static\_cast* ed è quindi da preferirsi nei casi in cui fosse consentito l'uso di entrambi.  
   Fermo restando che il cosiddetto *upcast*, ossia la conversione da classe erede a classe antenata, non richiederebbe alcun *casting*, come già si è più volte detto, trattandosi di una conversione implicita, *dynamic\_cast* si usa per il *downcast* (cosa che fa anche, in maniera INSICURA, *static\_cast*) e ANCHE per il cosiddetto *sidecast*, quando fosse possibile in modo non ambiguo e comunque SEMPRE con controllo sulla riuscita: se la conversione fallisse viene generato un puntatore nullo o lanciata un'eccezione, ossia viene eseguita un'espressione *throw*.  
   Che cosa sia il *downcast* è già stato detto, e del resto non è altro che il contrario dell'*upcast*; quanto al *sidecast* non si tratta di una motocicletta con tre ruote, ma della conversione tra puntatori a classi cugine, come del resto il nome lascia intuire.  
   Come sempre gli esempi illustreranno i concetti meglio delle parole: immaginate di avere per le mani una gerarchia di classi alquanto ramificata, sul tipo dell'albero genealogico dei paperi, come codificato da Don Rosa e che trovate a [questo indirizzo](http://www.physycom.unibo.it/labinfo/algoritmi/paperi.pdf).  
   Immaginate altresì che ogni icona di un papero rappresenti una classe e avrete una gerarchia come quella che abbiamo ipotizzato e che vi suggerisco anche di implementare in un codice, come esercizio: per renderla polimorfa, e quindi gestibile dall'operatore *dynamic\_cast*, sarebbe sufficiente inserire una funzione virtuale (come DOVRESTE SAPERE), anche del tutto inutile, nelle classi *Paperinocchio* e *Cornelius\_Coot*.   
   Ora, se si effettua la seguente dichiarazione di un puntatore a *Paperino*,  
     
   *Paperino \*paperino;*  
     
   questo puntatore, una volta adeguatamente inizializzato, può essere tranquillamente assegnato, per *upcast*, tanto al tipo *Ortensia\_de\_Paperoni* quanto a *Quackmore\_Duck*, ossia sono tranquillamente accettate entrambe le seguenti dichiarazioni inizializzate:   
     
   *Ortensia\_de\_Paperoni \* ortensia = paperino;  
   Quackmore\_Duck \* quackmore = paperino;*A questo punto come potrebbe essere inizializzato un puntatore a uno qualsiasi dei tre nipotini di Paperino? Certamente utilizzando l'operatore *new* e riservandosi direttamente della memoria nuova di zecca, ma forse anche riciclando memoria già utilizzata e tentando un *downcast*, con l'operatore *dynamic\_cast*, di uno dei due predetti puntatori. Usando *Quo* come campione si potrebbe provare con   
     
   *Quo \* quo = dynamic\_cast<Quo \*>(quackmore);*Il compilatore accetterà senza discutere questa dichiarazione inizializzata, perché, a parte svarioni clamorosi, come il non polimorfismo della gerarchia o il fatto che *Quo* NON SIA nella linea ereditaria di *quackmore*, sa benissimo che la liceità dell'operazione va verificata al tempo dell'esecuzione del codice.  
     
   A QUEL momento l'iniziativa del *downcast* presa dal programmatore viene frustrata, e la ragione è piuttosto chiara: il puntatore *quackmore*, essendo inizializzato correttamente tramite un puntatore a *Paperino*, punta l'oggetto della classe *Quackmore* ivi contenuto; ma il puntatore *paperino*, che ha bensì contezza di un oggetto *Quackmore* da far puntare a *quackmore*, NON NE HA ALCUNA di VERUN oggetto della classe *Della\_Duck* che invece si dovrebbe trovare nella linea ereditaria di *Quo* (guardate la figura). Il puntatore *quo* viene quindi inizializzato con un puntatore NULLO, in fase di esecuzione del programma, e se venisse usato SENZA PRIMA riscontrarne un valido valore, cagionerebbe, con probabilità prossima a uno, la catastrofe. Tutto quanto fin qui detto è verificabile immediatamente eseguendo il seguente codice, in cui si usano delle *struct* (dal nome abbreviato) SOLO PERCHÉ, nel presente contesto, NON È INTERESSANTE occuparsi di accessi pubblici o privati ai membri delle varie classi.

# include <iostream>  
using namespace std;  
  
struct Cornelius\_Coot  
{  
virtual void f(  ) {  };  
/\* per polimorfizzare il ramo americano \*/  
};  
  
struct Malcolm\_de\_Paperoni  
{  
virtual void f(  ) {  };  
/\* per polimorfizzare il ramo scozzese \*/  
};  
  
struct Ortensia : Malcolm\_de\_Paperoni  
{  
int ortensia;  
Ortensia(  ) : ortensia(11)  
{cout << "sono Ortensia\n";}  
};  
  
struct Quackmore : Cornelius\_Coot  
{   
int quackmore;  
Quackmore(  ) : quackmore(12)   
{cout << "sono Quackmore\n";}  
};  
  
struct Paperino : Ortensia, Quackmore  
{  
int paperino;  
Paperino(  ) : paperino(22)   
{cout << "sono Paperino\n";}  
};  
  
struct Della : Ortensia, Quackmore  
{  
int della;  
Della(  ) : della(23)  
{cout << "sono Della\n";}  
};  
  
struct Quo : Della  
{  
int quo;  
Quo(  ) : quo(34)  
{cout << "sono Quo\n";}  
};  
  
int main(  )  
{  
Paperino \* paperino = new Paperino;  
cout << "realizzato Paperino\n";  
Quackmore \* quackmore = paperino;  
Ortensia \* ortensia = paperino;  
cout <<  
"assegnato Paperino a Quackmore e Ortensia\n";  
cout << "a riprova di ciò:\n"   
<< ortensia -> ortensia << '\n'  
<< quackmore -> quackmore << '\n';  
Quo \* quo;  
try  
{  
quo = dynamic\_cast<Quo \*>(quackmore);  
cout <<  
"se appare questo, significa che non si lancia alcuna eccezione\n";  
}  
catch(...) // come da regolamento...  
{  }  
if(quo) cout << quo-> della << '\n';  
else cout <<  
"tuttavia il puntatore quo è NULLO!\n";  
}

Un *sidecast* riuscito potrebbe essere, ad esempio, la conversione di un puntatore a *Quackmore* in un puntatore a *Ortensia* in seguito a *upcast* di un puntatore a *Quo*, come mostra chiaramente il seguente codice, ripreso dal precedente e nel quale si ripassano in rassegna, illustrate tramite i commenti inseriti e i nomi stessi di classi e oggetti, alcune delle nozioni che dovreste già conoscere:

# include <iostream>  
using namespace std;  
  
struct Albero\_dei\_Paperi  
{virtual void astrazione(  ) = 0;  
/\* per polimorfizzare TUTTA la gerarchia  
stavolta a partire da una classe ASTRATTA \*/  
};  
  
struct Cornelius\_Coot  
: virtual Albero\_dei\_Paperi  
/\* per evitare il problema del diamante \*/  
{  
virtual void astrazione(  ) override {  };  
/\* per concretizzare il ramo americano \*/  
};  
  
struct Malcolm\_de\_Paperoni  
: virtual Albero\_dei\_Paperi  
/\* per evitare il problema del diamante \*/  
{  
virtual void astrazione(  ) override {  };  
/\* per concretizzare il ramo scozzese \*/  
};  
  
struct Ortensia : Malcolm\_de\_Paperoni  
{  
int ortensia;  
Ortensia(  ) : ortensia(11)  
{cout << "sono Ortensia\n";}  
};  
  
struct Quackmore : Cornelius\_Coot  
{   
int quackmore;  
Quackmore(  ) : quackmore(12)   
{cout << "sono Quackmore\n";}  
};  
  
struct Paperino : Ortensia, Quackmore  
{  
int paperino;  
virtual void astrazione(  ) override final {  }  
/\* per disambiguare il final overrider \*/  
Paperino(  ) : paperino(22)   
{cout << "sono Paperino\n";}  
};  
  
struct Della : Ortensia, Quackmore  
{  
int della;  
virtual void astrazione(  ) override final {  }  
/\* per disambiguare il final overrider \*/  
Della(  ) : della(23)  
{cout << "sono Della\n";}  
};  
  
struct Quo : Della  
{  
int quo;  
// il final overrider è quello di Della  
Quo(  ) : quo(34)  
{cout << "sono Quo\n";}  
};  
  
int main(  )  
{  
Paperino \* paperino = new Paperino;  
cout << "realizzato Paperino\n";  
Quackmore \* quackmore = paperino;  
Ortensia \* ortensia = paperino;  
cout <<  
"assegnato Paperino a Quackmore e Ortensia\n";  
cout << "a riprova di ciò:\n"   
<< ortensia -> ortensia << '\n'  
<< quackmore -> quackmore << '\n';  
Quo \* quo;  
try  
{  
quo = dynamic\_cast<Quo \*>(quackmore);  
cout <<  
"se appare questo, significa che non si lancia alcuna eccezione\n";  
}  
catch(...) // come da regolamento...  
{  }  
if(quo) cout << quo-> della << '\n';  
else cout <<  
"tuttavia il puntatore quo è NULLO!\n";  
// fin qui main è esattamente come prima,  
// poi si aggiunge  
quo = new Quo;  
Albero\_dei\_Paperi \* papero\_astratto = quo;  
quackmore =  
dynamic\_cast<Quackmore \*>(papero\_astratto);  
ortensia =  
dynamic\_cast<Ortensia \*>(quackmore);  
if(ortensia && quackmore) cout << "Bingo\n",  
cout << ortensia -> ortensia << '\n'  
<< quackmore -> quackmore << '\n';  
}

**Commenti al precedente elenco:**  
  
Spero che, a questo punto, risulti giustificata la premessa con cui questo canto ha esordito, e anche il titolo del canto stesso. Ricorrete al *casting* SOLO in caso di necessità e solo se non fosse già disponibile in versione implicita. Se componete delle vostre classi considerate se sia opportuno corredarle di operatori di *casting* definiti da voi stessi, piuttosto che ricorrere, il meno possibile, a quelli qui discussi. Soprattutto NON ILLUDETEVI di usare *const\_cast* per andare a modificare una variabile qualificata *const*: NON SERVE A QUESTO, ma solo a evitare errori di compilazione quando voleste trasmettere un puntatore a costanti a una funzione che non se le aspetta tali. Se poi la funzione prova a cambiarle, poco vi gioverà essere riusciti a compilare il programma, perché vi andrete a cacciare nel famigerato, e mai abbastanza temuto, ***undefined behaviour***, ossia in una di quelle deprecabili e nequitose situazioni, ancora peggiori del copia-incolla, in cui il calcolatore fa essenzialmente quello che vuole lui piuttosto che quello che vorreste voi.  
In simili frangenti, considerate una **benedizione celeste** il comunemente reietto Errore di segmentazione (o *Segmentation fault* che dir si voglia), perché, in sua mancanza, il rischio gravissimo che correrete è di coltivare l'illusione di avere scritto un programma perfettamente funzionante...che cadrà miseramente e con ignominia appena lo porterete su un altro calcolatore (tipicamente quello del vostro docente il giorno in cui verrete a sostenere l'esame: la legge di Murphy è spietata e senza misericordia).  
Utilizzate *reinterpret\_cast* SOLO quando sarete diventati grandi; per adesso SOLO nel contesto che vi è stato dato come esempio. E, se possibile, preferite *dynamic\_cast* a *static\_cast* quando vi trovaste costretti (ribadisco: costretti) a effettuare *casting* tra classi di una gerarchia: magari fatela diventare polimorfa apposta (ci vuole poco, come avete potuto notare dagli esempi).  
E se proprio vi dovesse servire *dynamic\_cast* NON DIMENTICATE di controllarne l'esito, altrimenti a che servirebbe il tempo che esso spreca per darvi questa facoltà?  
Ricordate, infine, che, per minimizzare i fallimenti di *dynamic\_cast*, occorre sempre usarlo *"cum grano salis"*, ossia evitando come la peste bubbonica percorsi eccessivamente tortuosi (vedi titolo) lungo la gerarchia di classi e, al contrario, voli troppo arditi tra la classe di partenza e quella di destinazione; abbiate cura di cautelarvi contro ogni possibile ambiguità che possa indurre l'operatore ad arrendersi, come eredità doppie e/o difetti nelle definizioni dei *final overriders*. In pratica conducete per mano l'operatore lungo la gerarchia in modo che sappia SEMPRE dove posare il piede che compie il passo successivo:  
  
*Poi ch'èi posato un poco il corpo lasso,  
ripresi via per la piaggia diserta,****sì che 'l pie' fermo sempre era 'l più basso.***

**Settima pausa di riflessione**

A questo punto del vostro percorso DOVRESTE SAPERE QUASI TUTTO QUELLO CHE **DOVETE SAPERE**. Domandatevi SERIAMENTE, come nella quinta pausa di riflessione, se è davvero così, e se la risposta dovesse essere meno che completamente positiva NON ESITATE a ricorrere all'interazione diretta col vostro docente, esternando ogni perplessità e ogni incomprensione (significa comunque che DOVETE essere passati per di qui, altrimenti facciamo a prenderci in giro).  
Manca ancora poco al raggiungimento della meta, e sarebbe davvero da scemi non riuscire ad arrivarci, o arrivarci morti, per non aver voluto o saputo fare domande.

**Canto trentaquattresimo: tutta la verità sulle inizializzazioni**

Questo è il primo canto che viene ascritto al livello altissimo, e tuttavia sarà un canto di tutto riposo, dedicato solo al riordino di cose in larghissima parte già viste e dette qua e là, al fine di mettere un punto fermo sul concetto di inizializzazione, di cui si parlò per la prima volta nel lontano canto secondo limitandosi allo stretto indispensabile a poter iniziare il cammino.  
Dopo quel canto introduttivo si sono incontrate molte forme diverse di inizializzazione: in questo canto verranno tutte compendiate e sistematicamente classificate, in modo da farne un momento di consultazione utile in ogni frangente. È come se gettassimo l'ancora della *piccioletta barca* approfittando di un giorno di bonaccia per rammendare le vele: torneranno a servire, entro breve, in tutta la loro efficienza.  
Oppure, in alternativa, visto che ormai siamo nel bel mezzo dell'*alto sale*, quindi l'ancora non si può gettare, né si vede uno straccio di lembo di terra entro tutti i trecentosessanta gradi dell'orizzonte, è come se ci riposassimo sotto coperta a ricordare i racconti che ci avevano fatto i vecchi marinai prima di salpare... e scoprissimo, adesso che si è fatta esperienza diretta dei fatti narrati nelle loro storie, che erano un po' *artefatti*, come sempre accade in quello che raccontano i vecchi marinai attorno al fuoco, specialmente quando fossero un tantino brilli.   
  
Dato il titolo del canto, è giunto il momento, per il vostro docente, vecchio marinaio anche lui, di confessare di aver abusato, a puro fin di bene, del termine inizializzazione quando venne utilizzato, nel canto secondo, per denominare ciò che, in verità, erano l'assegnamento e la lettura: per il rigore semantico di cui ORMAI siete capaci di portare il peso, occorre dire che il termine deve essere riservato SOLO ALLA PRIMA attribuzione di un valore a una variabile e quindi SOLO all'atto della COSTRUZIONE della variabile ovvero all'atto della sua DICHIARAZIONE. In altre parole...la verità è che NON ESISTONO DICHIARAZIONI che NON SIANO IMMEDIATAMENTE INIZIALIZZATE: piuttosto qualche immediata inizializzazione è peggiore di altre o addirittura pericolosa rispetto ad altre.  
  
Esistono dunque le seguenti forme di inizializzazione, NON UNA IN PIÙ NÉ UNA IN MENO:

1. inizializzazione cosiddetta *default*;
2. inizializzazione per valore;
3. inizializzazione per copia;
4. inizializzazione diretta;
5. inizializzazione aggregata;
6. inizializzazione con lista;
7. inizializzazione di un riferimento;
8. inizializzazione a zero;
9. inizializzazione costante;
10. inizializzazione dinamica non ordinata;
11. inizializzazione dinamica ordinata;
12. inizializzazione di membri mediante lista;
13. inizializzazione di membri *brace-or-equal*.

Denominando da ora, e per tutto il resto del canto presente, col termine *Tipo* per l'appunto un tipo qualsiasi, nativo o no, si darà adesso la spiegazione dettagliata di ognuna delle predette forme e vedrete che riconoscerete quasi sempre concetti già espressi. Salvo esplicito avviso diverso sarà sempre sottinteso che le dichiarazioni portate come esempi avvengano all'interno di un ambito di funzione, ossia NON nell'ambito di una classe o in un *namespace* e NEPPURE nell'ambito globale.

1. Quando si scrivono dichiarazioni come questa:  
     
   *Tipo variabile;*  
     
   o quando si usa l'operatore *new* in questo modo:  
     
   *Tipo \* puntatore;  
   puntatore = new Tipo;*si parla di inizializzazione *default* (si era chiamata questa una NON INIZIALIZZAZIONE); sappiamo già che, se *Tipo* è una classe, sia *variabile* sia l'oggetto puntato da *puntatore* sono inizializzati col costruttore (appunto) di *default*, mentre, se *Tipo* è nativo, non viene fatto proprio NIENTE, ed è in questo senso che si può affermare che, allora, gli oggetti coinvolti NON SONO INIZIALIZZATI.  
   Occorre aggiungere che l'inizializzazione *default* coinvolge OGNI SINGOLO elemento di un *array* dichiarato così:  
     
   *Tipo array[4];*e OGNI VARIABILE MEMBRO NON *static* che non appaia nella lista di inizializzazione del costruttore invocato per realizzare un oggetto della classe di appartenenza, compresa TUTTA l'eventuale classe antenata un cui costruttore diverso da quello di *default* NON SIA INSERITO nella citata lista di inizializzazione.  
   Qualche compilatore tratta diversamente l'inizializzazione *default* quando coinvolge le variabili locali di tipo nativo della funzione *main*, dirottandola sull'inizializzazione a zero (vedi appresso), MA NON SE NE PUÒ FARE UN DOGMA: del resto quegli stessi compilatori non applicano questa riconversione né alle variabili locali native dichiarate in ambiti di altre funzioni né a quelle dichiarate in ambiti di classi.  
   L'inizializzazione *default* NON SI PUÒ APPLICARE, pena segnalazione di errore in compilazione, nei seguenti casi:
   * nella dichiarazione di un riferimento, sinistro o destro che sia, come in:   
       
     *Tipo & riferimento\_sinistro; // ERRORE  
     Tipo && riferimento\_destro; // ERRORE*(a meno che non si tratti della dichiarazione di un MEMBRO di una classe);
   * nella dichiarazione di un oggetto qualificato *const* se *Tipo* non è una classe provvista di un costruttore *default* **fornito dal programmatore** come in:  
       
     *const int variabile; // ERRORE*
2. L'inizializzazione per valore si ha quando si scrivono dichiarazioni come questa:   
     
   *Tipo variabile{  }; // una coppia di graffe vuota***ovvero** ogni volta che in un'espressione appare un oggetto senza nome a causa della presenza di sottoespressioni, precedute o no dall'operatore *new*, come *Tipo(  )* oppure *Tipo{  }*, vale a dire col nome del tipo seguito da una coppia vuota di parentesi tonde o graffe,  
   **ovvero** quando nella lista di inizializzazione di un costruttore compare il nome di una variabile membro NON *static* seguito da una coppia vuota di parentesi tonde o graffe.  
     
   Gli effetti di questo genere di inizializzazione, forse una delle poche novità di questo canto, dipendono da che cosa sia *Tipo* e da quali parentesi si usano:
   * se *Tipo* è una classe, sono usate le parentesi graffe, e la classe NON HA un costruttore di *default*, MA È FORNITA di un costruttore che riceve per argomento un oggetto di tipo *initializer\_list*, l'inizializzazione per valore è dirottata sull'inizializzazione con lista (vedi appresso);
   * se *Tipo* è una classe, sono usate le parentesi graffe, e la classe HA un costruttore di *default* fornito dal programmatore, è utilizzato QUESTO COSTRUTTORE. Se il costruttore di *default* è quello implicito ci si dirotta sull'inizializzazione a zero;
   * se *Tipo* è una classe, sono usate le parentesi graffe, e la classe NON HA né un costruttore di *default* né un costruttore che riceva per argomento un oggetto di tipo *initializer\_list*, SI GENERA UN ERRORE IN COMPILAZIONE;
   * se *Tipo* è una classe e sono usate le parentesi tonde, significa che NON CI SI TROVA AL COSPETTO DI UNA DICHIARAZIONE, ma in uno dei due "**ovvero**" citati sopra. In tal caso:
     1. se la classe è fornita di un costruttore qualsiasi, viene richiesto e invocato il costruttore di *default*; se questo non è, a sua volta, disponibile, o perché anch'esso definito dal programmatore o perché ricuperato tramite la parola di vocabolario *default*, si causa un errore di compilazione;
     2. se la classe NON HA UNO STRACCIO DI COSTRUTTORE fornito, l'inizializzazione si dirotta sull'inizializzazione a zero (vedi appresso);
   * se *Tipo* è nativo, e si usano le parentesi graffe, ci si dirotta sull'inizializzazione a zero (vedi appresso);
   * se *Tipo* è nativo, e si usano le parentesi tonde, ancora una volta significa che ci si trova in un "**ovvero**" e NON IN UNA DICHIARAZIONE, ma anche in questo caso l'oggetto senza nome prodotto ricade nell'inizializzazione a zero;
   * infine, se *Tipo* è un aggregato (ricordate questo termine? Anch'esso era stato usato impropriamente quando ci si accostava con timidezza agli oggetti: qui viene usato nella sua propria semantica, per la quale si rimanda qui appresso, allorché si descriverà appunto l'inizializzazione aggregata), allora subisce inizializzazione aggregata (vedi appresso) se si usano le graffe, mentre se si usano le tonde OGNI elemento dell'aggregato subisce inizializzazione per valore, e l'esito è pertanto già scritto in questo stesso elenco, secondo quale sia la natura dei singoli elementi dell'aggregato.

Occorre che vi si spieghi il motivo per cui, quando si usano le parentesi tonde, NON CI SI PUÒ TROVARE DI FRONTE A UNA DICHIARAZIONE da dover inizializzare? Tornando all'inizio di QUESTO PUNTO dell'elenco, se immaginaste di sostituire la coppia vuota di graffe con una coppia vuota di tonde... che cosa avreste ottenuto?

1. L'inizializzazione per copia è una delle nostre vecchie conoscenze più care; avviene quando una dichiarazione appare così:  
     
   *Tipo variabile = espressione;*   
     
   è quella che avevamo sempre chiamato, semplicisticamente, IMMEDIATA INIZIALIZZAZIONE. Ora comprendiamo che, secondo la natura di *Tipo* e secondo com'è fatta *espressione*, possono essere implicate azioni sottintese MOLTO diverse:
   * se *espressione* è riconducibile a un tipo nativo, e anche *Tipo* è tale, non dovrebbero occorrere altre parole;
   * altrimenti si cerca uno adatto tra i costruttori di *Tipo* (ESCLUSI QUELLI QUALIFICATI CON LA PAROLA DI VOCABOLARIO *explicit*) o fra gli operatori di *casting* della classe di *espressione* forniti dal programmatore e lo si invoca;
   * se non se ne trova neanche uno, si tratta di un errore in compilazione.

Va notato che, se *espressione* è un *rvalue*, nella ricerca dei costruttori il *move constructor* (se presente) prevale sul *copy constructor*; si tornerà su questo dettaglio.  
Va altresì notato, anche se non dovrebbe essercene bisogno, che l'inizializzazione per copia che qui si discute NON HA NULLA DA SPARTIRE con qualsiasi eventuale *overload* dell'operatore di assegnamento.  
L'inizializzazione per copia avviene anche, tacitamente, ma EFFICACEMENTE:

* + a partire da ogni parametro trasmesso a una funzione che corrisponda a un argomento ricevuto per valore;
  + teoricamente a partire dall'espressione a destra di *return*, dall'interno di una funzione, verso il chiamante, anche se quasi tutti i compilatori applicano, in questo caso, la cosiddetta *return value optimization*, evitando quindi QUESTA inizializzazione;
  + a partire dall'espressione a destra di *throw*, verso un *catch* che la riceva, ancora una volta, per valore.

1. Anche l'inizializzazione diretta è qualcosa di ormai abituale, anche al vostro livello di principianza. Avviene quando una dichiarazione ha questo aspetto:  
     
   *Tipo variabile(espressione\_1, espressione\_2, espressione\_etcetera);*  
     
   vale a dire una forma del tutto simile a quella introdotta nell'inizializzazione per valore citata al punto 2., ma con la coppia di parentesi NON VUOTA; osservate che stavolta si tratta di una VALIDA DICHIARAZIONE  
   **ovvero** quando una sottoespressione, preceduta o no dall'operatore *new*, del genere *Tipo(espressione\_1, espressione\_2, espressione\_etcetera)* compare in un'espressione,  
   **ovvero** quando il nome di una variabile membro NON *static*, seguito da *(espressione\_1, espressione\_2, espressione\_etcetera)* compare nella lista di inizializzazione di un costruttore,  
   **ovvero** per inizializzare l'oggetto innominato di tipo *Tipo* prodotto dall'operatore *static\_cast*,  
   **ovvero** per inizializzare le variabili catturate da un'espressione *lambda* (quest'utima è da rinviare al futuro).  
     
   Evidentemente si tratta della richiesta invocazione di un appropriato costruttore, COMPRESI quelli qualificati *explicit*, quando *Tipo* è una classe o dell'applicazione dell'appropriato *casting* implicito quando *Tipo* è nativo.  
   In difetto dell'esistenza di un costruttore idoneo (ad esempio se si inserisce tra parentesi più di UNA SOLA espressione quando *Tipo* è nativo) si genera un errore di compilazione.
2. L'inizializzazione aggregata è riservata appunto agli aggregati; con questo termine si intendono:
   * gli *array*, sia allocati staticamente, sia creati con l'operatore *new*;
   * le *struct* e le *union* dell'antico C-ANSI, vale a dire classi, anche del linguaggio C++, che siano del tutto sprovviste di:
     1. qualsiasi membro NON pubblico;
     2. qualsiasi costruttore fornito dal programmatore;
     3. qualsiasi classe da cui derivino;
     4. anche UN SOLO metodo virtuale;
     5. anche UN SOLO membro inizializzato con una sintassi che usi le graffe.

Si attua in uno di questi due modi:  
  
*Tipo aggregato\_1 = {espressione\_1, espressione\_2, espressione\_etc};  
Tipo aggregato\_2 {espressione\_1, espressione\_2, espressione\_etc};*  
  
Riconoscerete anche adesso cose già viste; va notato che se l'aggregato da inizializzare è una *union*, entro le graffe va posta UNA SOLA espressione, qualunque sia la complessità della *union* stessa; tale unica espressione servirà a inizializzare il PRIMO MEMBRO della *union*.  
L'inizializzazione aggregata, in ultima analisi, non è altro che un'inizializzazione per copia (già discussa) di OGNI ELEMENTO dell'aggregato, fatta per ordine di comparizione nell'aggregato stesso e nelle parentesi graffe inizializzanti.  
SI PRODUCE UN ERRORE DI COMPILAZIONE se il numero di inizializzatori è MAGGIORE del numero di elementi dell'aggregato, mentre se è MINORE gli elementi residui subiscono inizializzazione per valore (ossia quello che avverrebbe con le graffe vuote, ricordate?). Ne segue che, se tra codesti elementi residui ve n'è anche SOLO UNO che sia un riferimento, il quale NON PUÒ essere inizializzato per valore, si produce un ERRORE di compilazione.  
È consentita la presenza nelle graffe di inizializzatori che siano espressioni della categoria *rvalue*, ma in tal caso, dato che l'inizializzazione per copia dell'elemento coinvolto potrebbe richiedere *casting* impliciti, SONO VIETATE, a prezzo di ERRORE di compilazione, le cosiddette conversioni *narrowing*, che generalmente sarebbero permesse, se il programmatore ne fosse consapevole, ossia quelle che implicano perdita di bit: per esempio un *int* che dovesse diventare uno *short int*; non si consente di infilare un piede 42 in una scarpa 38, neanche se uno si dichiarasse disponibile alla più atroce sofferenza.  
Si è già visto molte volte che, se l'aggregato è un *array* statico, l'omissione della sua estensione è compensata dal conteggio del numero di inizializzatori, mentre che ciò non sia possibile quando a creare l'*array* è l'operatore *new* è dovuto semplicemente al fatto che l'inizializzazione aggregata NON precede la creazione dell'*array* e quindi BISOGNA che *new* sappia comunque quanta memoria allocare.  
Altresì si è già visto che all'interno delle graffe possono trovarsi nidificati altri inizializzatori di aggregati: ciò è del tutto naturale, sia per *array* multi-indice sia per strutture che abbiano come membri altre strutture; si è anche visto che per tali inizializzatori nidificati si possono anche omettere le parentesi graffe che li delimiterebbero, ma SOLO SE SI ADOTTA LA SINTASSI COL SEGNO = (uguale). Pare, tuttavia, che questo diverso comportamento possa essere abolito in un prossimo futuro (chi vivrà vedrà). D'altra parte dovrebbe apparire scontato che una simile abolizione delle parentesi graffe relative a inizializzatori nidificati non possa applicarsi quando una struttura avesse come membro un'altra struttura VUOTA: in un caso simile è ovvio che debba comparire tra le graffe una coppia di graffe VUOTA, altrimenti si genererebbe una violazione d'accesso o comunque uno sparigliamento degli inizializzatori. Resterebbe da chiedersi che cosa ci stia a fare una struttura vuota dentro un'altra struttura, ma qui vuota vuol significare priva di membri inizializzabili, NON VUOTA AFFATTO: e una struttura può tranquillamente essere priva di membri inizializzabili, senza essere vuota, se si pensa che durante il processo di inizializzazione aggregata sono comunque saltati, vale a dire IGNORATI, TUTTI gli eventuali membri *static* e TUTTI gli eventuali *bitfields*.  
L'ultima cosa da dire è anch'essa nota da tempo, ossia che quando l'aggregato è un *array* del tipo *char*, in tutte le sue possibili modificazioni, l'inizializzazione aggregata può avvenire anche tramite una stringa delimitata da virgolette e che potrebbe anche essere racchiusa tra graffe. Tale inizializzazione inserisce automaticamente il *byte* nullo come ultimo elemento dell'aggregato, che quindi deve avere capienza sufficiente o lasciata calcolare al compilatore.

1. L'inizializzazione con lista, nata nello standard 2011 del linguaggio, richiede la conoscenza, almeno approssimativa, di una classe *template*, definita nel *namespace std*, e la cui dichiarazione suona  
     
   *template <class tipo\_degli\_inizializzatori> class initializer\_list;*   
     
   Si tratta di un contenitore di oggetti di tipo *tipo\_degli\_inizializzatori*, in quantità non preventivamente specificata, una cui istanza viene realizzata automaticamente dal compilatore appunto quando viene riscontrato nel codice un elenco di espressioni di ugual tipo separato da virgole e racchiuso tra graffe, in ogni contesto in cui un oggetto di tal fatta possa comparire. Pertanto una inizializzazione con lista è quasi per nulla diversa da un'inizializzazione per copia o un'inizializzazione diretta in cui il tipo dell'UNICA espressione inizializzante sia proprio  
   *initializer\_list<tipo\_degli\_inizializzatori>*.  
   I ragionamenti compiuti dal compilatore, posto davanti a una dichiarazione come questa, o davanti a una sottoespressione in cui *Tipo* sia immediatamente seguito da un'istanza di *initializer\_list*, o in qualunque altra circostanza equivalente  
     
   *Tipo oggetto{espressione1, espressione2, altre\_espressioni};*   
     
   si dipanano secondo il seguente schema:
   * se la coppia di graffe è vuota, e *Tipo* è una classe dotata di costruttore *default*, *oggetto* è inizializzato tramite tale costruttore;
   * se la coppia di graffe non è vuota e *Tipo* è un aggregato, *oggetto* subisce l'inizializzazione aggregata;
   * se nessuno dei punti precedenti si è potuto concretizzare e *Tipo* è una classe, si vanno a spulciare i suoi costruttori alla ricerca di, nell'ordine:
     1. un costruttore con un unico argomento, al netto di argomenti *standard*, di tipo *initializer\_list<tipo\_congruo>*, ove *tipo\_congruo* si autocommenta; se viene trovato lo si utilizza per inizializzare *oggetto*;
     2. se un tale costruttore non viene trovato, si cerca, tra tutti gli altri costruttori, se ve ne sia uno con una lista di argomenti combaciante, al lordo di conversioni implicite che non implichino restringimento di *bit*, col contenuto ordinato dell'oggetto *initializer\_list* inizializzante: se un tale costruttore non viene trovato SI GENERA UN ERRORE DI COMPILAZIONE; altrimenti QUEL miglior costruttore è usato per eseguire l'inizializzazione diretta di *oggetto*. È essenziale che non venga usato il segno = perché se lo si usa si ha un'inizializzazione per copia (cfr. sopra) e non con lista: in tal caso, pur se esistesse un costruttore adeguato, ma fosse qualificato *explicit*, l'errore di compilazione si produrrebbe ugualmente;
   * nel caso particolare in cui all'interno della coppia di graffe ci sia UN SOLO inizializzatore, si procede subito come per una inizializzazione diretta o per copia, secondo l'istanza di *initializer\_list*, a partire dal tipo dell'unico inizializzatore presente;
   * infine, se *Tipo* è un riferimento, viene creato sull'istante un oggetto innominato temporaneo del tipo *Tipo*, che subisce inizializzazione con lista esattamente secondo quanto detto fin qui e, se l'inizializzazione ha buon fine, il riferimento dichiarando viene appunto riferito a tale oggetto temporaneo innominato (si tornerà su questo, vedi anche appresso).
2. L'inizializzazione di un riferimento serve appunto a collegare fra loro un riferimento e l'oggetto cui si riferisce; quando un riferimento viene dichiarato NECESSITA SEMPRE di QUESTO tipo di inizializzazione che finisce per essere una sua esclusiva (a meno che non sia dichiarato nell'ambito globale e anche qualificato *extern*...  
   ma questa è DAVVERO una pignolata delle mie).  
   Occorre distinguere fra riferimenti sinistri (*lvalue references*) e riferimenti destri (*rvalue references*), che sono stati citati spesso nel percorso senza mai addentrarsi nelle profondità di significato di questi aggettivi, non si sa se di natura politica o solo topologica.  
   Non lo faremo neanche adesso, rimandando la patata bollente a un appropriato canto, e ci limiteremo a dare le possibili forme di inizializzazione senza ancora spiegarle troppo. Detto che le inizializzazioni di un riferimento destro sono identiche a quelle di un riferimento sinistro, distinguendosene solo per l'uso del doppio segno && al posto di un singolo &, proprio per questa ragione ci si limiterà a citare le ultime. Avremo quindi queste tre possibili inizializzazioni:  
     
   *Tipo & riferimento\_1 = oggetto;  
   Tipo & riferimento\_2 (oggetto);  
   Tipo & riferimento\_3 {oggetto};*   
     
   in cui appaiono dichiarati tre riferimenti sinistri diversi allo stesso *oggetto* con sintassi paragonabili rispettivamente a una inizializzazione per copia, una inizializzazione diretta e una inizializzazione con lista. Va tuttavia ben tenuto presente che qui si tratta di tutt'altro, dato che un riferimento NON È UN OGGETTO e quindi MAI SI SUPPONGA che queste inizializzazioni implichino la ricerca e l'esecuzione di qualsiasi costruttore, neppure quando *Tipo* sia una classe. A proposito di tipi, quello di *oggetto* DEVE ESSERE ESATTAMENTE *Tipo*, oppure, ma solo quando *Tipo* fosse una classe, una classe erede di *Tipo*; in quest'ultimo caso i riferimenti dichiarati sopra sono riferiti (si perdoni il bisticco) al sotto-oggetto di tipo *Tipo* (si perdoni ancora un altro bisticcio) che si trova in *oggetto* (ed ecco il terzo bisticcio).   
   Per la verità, e per completezza, il tipo di *oggetto* potrebbe anche non appartenere alla gerarchia ereditaria di *Tipo*, MA ALLORA, PER RENDERE PLAUSIBILI LE INIZIALIZZAZIONI PROPOSTE, BISOGNA che sia equipaggiato con funzioni di conversione a *Tipo* o a una classe erede di *Tipo* e BISOGNA ALTRESÌ che tali funzioni di conversione RESTITUISCANO APPUNTO UN RIFERIMENTO APPROPRIATO.  
   In ogni caso il tipo di *oggetto* deve essere NON PIÙ QUALIFICATO tramite *const* e/o *volatile* rispetto a quanto lo sia *Tipo*, pena errore di compilazione.  
   Le stesse inizializzazioni sono compiute tacitamente, ma EFFICACEMENTE, dal compilatore stesso (ricordate il punto 3. di questo stesso elenco?) allorché:
   * si trasmette un oggetto come parametro a una funzione che riceva come argomento un riferimento al tipo dell'oggetto;
   * una funzione restituisce al chiamante un riferimento.

Da ultimo va osservato che quando un riferimento si riferisce a un oggetto temporaneo innominato, quest'ultimo prolunga la sua vita fino a quando vive il riferimento che gli si riferisce, con poche eccezioni che concernono essenzialmente quelli erroneamente fatti restituire da funzioni che dovrebbero restituire un riferimento.

1. L'inizializzazione a zero riguarda in prima persona (finalmente) le dichiarazioni qualificate *static* (ricordate che finora si era sempre esclusa tale qualifica?), come ad esempio la dichiarazione  
     
   *static Tipo variabile;*in aggiunta a tutte le ricadute in questa forma di inizializzazione provenienti dalle altre forme, nei casi contemplati e discussi nei punti precedenti di questo elenco. Consiste semplicemente nello spegnimento di TUTTI i bit allocati in memoria per l'oggetto dichiarando: ne segue che, quando *Tipo* è un tipo numerico nativo, il valore assunto è quello della costante numerica 0 di quel tipo e la stessa cosa avviene per tutte le variabili membro di tipo numerico di una classe così inizializzata...etcetera...
2. L'inizializzazione costante riguarda anch'essa variabili/oggetti qualificati *static*, compresi anche i riferimenti SINISTRI così qualificati; ha la forma  
     
   *static Tipo oggetto\_o\_riferimento\_sinistro = espressione\_costante;*   
     
   ove con *espressione\_costante* si intende qualsiasi espressione cui il compilatore possa riconoscere l'applicabilità della qualifica *constexpr*, come stabilisce la grammatica del linguaggio (non andate, ADESSO, a compulsare la descrizione di tale parola di vocabolario); se *oggetto\_o\_riferimento\_sinistro* è un riferimento, allora *espressione\_costante* deve essere un *lvalue* a sua volta qualificato *static* oppure un *xvalue* temporaneo (soprassedere fino a nuovo ordine).  
   Serve essenzialmente, quando possibile, ad attribuire valori iniziali diversi da quello nullo dato dall'inizializzazione a zero, e ha per giunta la garanzia di essere eseguita in fase di compilazione E PRIMA di ognuna delle inizializzazioni dinamiche discusse appresso, anche quando queste comparissero in posizioni precedenti nel codice sorgente.
3. L'inizializzazione dinamica non ordinata riguarda SOLO i membri qualificati *static* di classi templatizzate NON esplicitamente specializzate; va da sé, almeno per il vostro livello, che accada alquanto di rado e consiste nell'applicare a tali membri una delle inizializzazioni fin qui descritte, secondo il contesto, ma senza alcun ordine preciso, e quindi si deve rifuggire da dipendenze ordinali di simili membri statici l'uno dall'altro: l'unica garanzia che si ha è che TUTTE queste inizializzazioni saranno completate PRIMA che inizi l'esecuzione della funzione *main* (e ci sarebbe mancato altro...).
4. L'inizializzazione dinamica ORDINATA, invece, come assicura l'aggettivo, avviene seguendo l'ordine delle dichiarazioni nel documento sorgente e consiste nell'applicare l'inizializzazione dovuta, tra quelle descritte, alle variabili/oggetti dichiarate nell'ambito GLOBALE, tenendo BEN PRESENTE che IVI NON TUTTE LE FORME DI INIZIALIZZAZIONE sono ammissibili. Ovviamente è garantito che siano TUTTE compiute PRIMA dell'inizio dell'esecuzione di *main*.
5. L'inizializzazione di variabili/oggetti membri mediante lista non è altro che quella che avviene tramite la lista di inizializzazione opzionale posta nella definizione di un costruttore, e della quale si è sovente parlato in questo stesso documento.
6. Infine, l'inizializzaziione *brace-or-equal* di variabili/oggetti membri NON *static*, che è stata introdotta nello standard 2011 del linguaggio, consente di attribuire valori iniziali ai membri di una classe nell'ambito della sua stessa definizione; le parole *brace-or-equal* significano che tali inizializzazioni possono attuarsi sia tramite il segno =, come nell'inizializzazione per copia, sia tramite una coppia di graffe, come nell'inizializzazione aggregata. Va però sottolineato che queste inizializzazioni **SONO IGNORATE** se il costruttore usato per istanziare l'oggetto inizializza a sua volta le stesse variabili membro.

Esempi a sfinimento (**ESEGUITELI TUTTI!**)

1. oltre a quelli già dati nell'elenco...
   * *class Ciccio {  
     int i; // subisce inizializzazione default  
     public:  
     Ciccio(  ) {  }  
     int I(  ) const {return i;}  
     };  
       
     int main(  ) {  
     const Ciccio c;  
     // inizializzazione default consentita  
     // perché Ciccio HA un costruttore default  
     // fornito dal programmatore  
     Ciccio d;  
     std::cout << c . I(  )<< '\n';  
     std::cout << d . I(  )<< '\n';  
     }*
   * *class Ciccio {  
     public:  
     Ciccio(std::initializer\_list<int> lista) {i = 1;}  
     int i;  
     };  
     // Ciccio NON HA un costruttore default  
     // ma ne HA uno che riceve un argomento std::initializer\_list  
       
     int main(  ) {  
     Ciccio c{  };  
     std::cout << c.i << '\n'; // c.i vale 1  
     }*
   * *class Ciccio {  
     public:  
     Ciccio(std::initializer\_list<int> lista) {i = 1;}  
     Ciccio(  ) {i = 2;}  
     int i;  
     };  
     // Ciccio ORA HA ANCHE un costruttore default  
     // oltre a quello di prima   
       
     int main(  ) {  
     Ciccio c{  };  
     std::cout << c.i << '\n'; // c.i vale 2 (!)  
     }*
   * *class Ciccio {  
     public:  
     int i;  
     };  
     // Ciccio ORA HA SOLO il costruttore default  
     // implicito: inizializzazione a zero   
       
     int main(  ) {  
     Ciccio c{  };  
     std::cout << c.i << '\n'; // c.i vale 0 (!!)  
     }*
   * *class Ciccio {  
     public:  
     Ciccio(int k) {i = k;}  
     int i;  
     };  
     // Ciccio ORA NON HA né il costruttore default  
     // né il costruttore initializer\_list   
       
     int main(  ) {  
     Ciccio c{  }; // ERRORE  
     std::cout << c.i << '\n';  
     }*
   * *class Ciccio {  
     public:  
     Ciccio(  ) {i=1;}  
     int operator+(Ciccio c)  
     {return i+c.i;}  
     int i;  
     };  
       
     template <typename X> void funza(X a)  
     {std :: cout << a + X(  ) << '\n';}  
     // se istanziata con Ciccio esegue operator+ e usa  
     // il costruttore default per l'operando X(  )  
     // se istanziata con tipo nativo inizializza a zero tale  
     // operando.  
       
       
     int main(  ) {  
     Ciccio c;  
     // segue inizializzazione per valore di ogni elemento  
     // di un aggregato di oggetti di una classe  
     Ciccio \*cc = new Ciccio[3](  );  
     // stesso effetto di cui sopra, finché Ciccio dispone  
     // di un costruttore default (fornito o implicito)  
     // SENZA un costruttore initializer\_list (cfr. esempi sopra)  
     Ciccio \*cc\_lo\_stesso = new Ciccio[3]{  };  
     // segue inizializzazione per valore di ogni elemento  
     // di un aggregato di oggetti di tipo nativo  
     int \* iii = new int[3](  );   
     // segue stesso effetto per tipo nativo  
     int \* iii\_lo\_stesso = new int[3]{  };   
     // NON SI PUÒ usare, per questa inizializzazione,  
     // un aggregato costituito da un array allocato staticamente  
     // int array\_erroneo[3](  ); // ERRORE!  
     // A MENO CHE NON SI INIZIALIZZI CON LE GRAFFE  
     int array\_corretto[3]{  };   
     int k{  }; // inizializzazione a zero  
     funza(c);  
     funza(k);  
     for(int i=0; i < 3; ++i) std::cout << (cc+i) -> i << ' ';  
     std::cout << '\n';  
     for(int i=0; i < 3; ++i) std::cout << (cc\_lo\_stesso+i) -> i << ' ';  
     std::cout << '\n';  
     for(int i=0; i < 3; ++i) std::cout << iii[i] << ' ';  
     std::cout << '\n';  
     for(int i=0; i < 3; ++i) std::cout << iii\_lo\_stesso[i] << ' ';  
     std::cout << '\n';  
     for(int i=0; i < 3; ++i) std::cout << array\_corretto[i] << ' ';  
     std::cout << '\n';  
     }*
2. esempi sono già stati dati lungo tutto il percorso; si ritiene solo utile sottolineare qualche dettaglio, correlato al punto precedente, di cui si riprende l'ultimo esempio, in una versione molto semplificata, e in cui la classe *Ciccio* viene equipaggiata di un costruttore di copia un po' baro, giusto per rendere evidente quando avviene inizializzazione per copia e quando no:   
     
   *class Ciccio {  
   public:  
   Ciccio(  ) {i=1;}  
   Ciccio(const Ciccio& c) {i = c.i + 3;}  
   int operator+(Ciccio c)  
   {return i+c.i;}  
   int i;  
   };  
     
   template <typename X> void funza(X a)  
   {X b{  };  
   std :: cout << a + X(  ) << '\n';  
   std :: cout << a + b << '\n';}  
     
     
   int main(  ) {  
   Ciccio c;  
   int k{  };   
   funza(c);  
   funza(k);  
   }*L'esempio dovrebbe spiegare adeguatamente quanto segue:
   * quando *main* richiede l'esecuzione di *funza(c)*, l'argomento di questa (*X a*, ossia *Ciccio a*) viene inizializzato per copia, come diffusamente spiegato nell'elenco puntato; e siccome il costruttore di copia di *Ciccio* è fornito, viene inizializzato tramite questo. Tale costruttore, lo si è anticipato, si comporta in modo un po' truffaldino: non esegue una copia fedele, ma una in cui la variabile membro *i* è incrementata di 3: al compilatore non importa un fico secco. Per questa ragione l'oggetto *a* in *funza* vale 4, ossia è questo il contributo che fornisce al proprio *operator+* entrambe le volte che lo invoca. L'altro contributo a *operator+* proviene, la prima volta, da un oggetto innominato temporaneo della classe *X*, ossia *Ciccio*, il quale, come pure spiegato nell'elenco, deve essere inizializzato per valore, vale a dire, essendo fornito, attraverso il costruttore di *default* di *Ciccio*; la seconda volta, diversamente, il contributo a *operator+* proviene da un autentico oggetto della classe *Ciccio*, già istanziato in precedenza nell'ambito di *funza*, e quindi, sempre secondo l'elenco, l'argomento di *operator+* deve essere inizializzato per copia. Ecco perché si vedono apparire due risultati diversi (5 e 8, rispettivamente) per le due operazioni: avete toccato con mano, forse per la prima volta, una delle differenze che passano tra un *rvalue* (*Ciccio(  )*) e un *lvalue* (*b*) usati uno al posto dell'altro in espressioni IDENTICHE. Evidentemente, se il costruttore di copia fosse stato onesto, questa differenza, PUR PRESENTE, non si sarebbe potuta apprezzare.
   * quando invece viene invocata *funza(k)* nulla di quanto detto sopra accade di nuovo: ogni intero coinvolto subisce sempre e comunque inizializzazione a zero e i due risultati coincidono.
3. di questo tipo di inizializzazione appare superfluo fornire esempi...o no? (delle espressioni *lambda* si è detto che si parlerà in futuro);
4. anche di questa forma sono stati dati esempi lungo il percorso;
5. in questo esempio ritroverete anche cose che vi dovrebbero essere già ampiamente note:  
     
   *#include <iostream>  
   #define MASSIMO 60  
   class Ciccio {  
   public:  
   Ciccio(std :: initializer\_list<int> lista)  
   {int k = -1;  
   size\_t s = lista . size(  );  
   if(s > MASSIMO)  
   std::cerr <<  
   "initializer\_list eccessiva:\nne saranno eliminati gli ultimi\n"  
   << s - MASSIMO << " elementi.\n";  
   for(int l : lista) {i[++k] = l; if(k == MASSIMO-1) break;}  
   quanti = k+1;  
   }  
   int i[MASSIMO], quanti;  
   };  
     
   class Cuccio  
   {double d[3];  
   public:  
   Cuccio(double d1, double d2, double d3)  
   {d[0] = d1, d[1] = d2, d[2] = d3;}  
   double \* D(  )  
   {return static\_cast<double \*>(d);}  
   };  
     
   class Caccio  
   {double d[3];  
   public:  
   explicit Caccio(double d1, double d2, double d3)  
   {d[0] = d1, d[1] = d2, d[2] = d3;}  
   double \* D(  )  
   {return static\_cast<double \*>(d);}  
   };  
     
   class Coccio  
   {int d[3];  
   public:  
   Coccio(int d1, int d2, int d3)  
   {d[0] = d1, d[1] = d2, d[2] = d3;}  
   int \* D(  )  
   {return static\_cast<int \*>(d);}  
   };  
     
   int main( ) {  
   Ciccio c{1, 2, 3, 4, 11, 21, 33, 66, -2, -4};  
   std::cout << "c ha dentro di sé " << c.quanti << " interi di valore\n";  
   for(int i=0; i < c.quanti; ++i) std::cout << c.i[i] << ' ';  
   std :: cout << '\n';  
   Cuccio cu{1.7, 2.1, 3.0};  
   double \*d = cu . D(  );  
   std :: cout << "ecco i valori contenuti in cu:\n"  
   << d[0] << ' '  
   << d[1] << ' '  
   << d[2] << '\n';  
   Caccio ca{1.7, 2.1, 3.0}; // ACCETTATO: inizializzazione con lista  
   double \*D = ca.D(  );  
   std :: cout << "ecco i valori contenuti in ca:\n"  
   << D[0] << ' '  
   << D[1] << ' '  
   << D[2] << '\n';  
   // Caccio caa = {1.7, 2.1, 3.0}; // ERRORE: inizializzazione per copia...  
   // ... da costruttore explicit  
   // Coccio co{1.7, 2.1, 3.0}; ERRORE: narrowing!  
   Coccio co(1.7, 2.1, 3.0); // ACCETTATO: inizializzazione diretta!  
   int \* i = co. D(  );  
   std :: cout << "ecco i valori contenuti in co:\n"  
   << i[0] << ' '  
   << i[1] << ' '  
   << i[2] << '\n';  
   Coccio && co\_rref{11, 22, 33}; // riferimento destro a oggetto temporaneo...  
   // ... innominato.  
   int \* i\_rref = co\_rref.D(  );  
   std :: cout << "ecco i valori riferiti da co\_rref:\n"  
   << i\_rref[0] << ' '  
   << i\_rref[1] << ' '  
   << i\_rref[2] << '\n';  
   }*
6. questo esempio, contenendo SOLO dichiarazioni, è completamente inutile, ma, essendo compilabile, se non si tolgono i commenti messi APPOSTA, illustra le possibili dichiarazioni inizializzate per i riferimenti:  
     
   *# include <iostream>  
     
   class Ciccio  
   {  };  
     
   class Pappo : public Ciccio  
   {  };  
     
   class Peppo  
   {  
   public:  
   operator Ciccio&(  )  
   {  
   return reinterpret\_cast<Ciccio &>(\*this);  
   }  
   };  
     
   int main(  )  
   {  
   Ciccio ciccio;  
   Pappo pappo;  
   Peppo peppo;  
   const Pappo pappino;  
   volatile Pappo pappetto;  
   const volatile Pappo pappuccio;  
   Ciccio &r\_ciccio = ciccio;  
   Ciccio &r\_pappo = pappo; // OK! pappo erede  
   const Ciccio & const\_ciccio = ciccio; // OK! const\_ciccio  
   // ...... più qualificato  
   // Pappo & r\_pappino = pappino; // ERRORE! pappino è const  
   // Pappo & r\_pappetto = pappetto; // ERRORE! pappetto è volatile  
   // const Pappo & r\_pappuccio = pappuccio; // ERRORE! pappuccio   
   // ...... è const volatile  
   // volatile Pappo &v\_pappino = pappino; // ERRORE! vedi sopra  
   // const Pappo &c\_pappetto = pappetto; // ERRORE! come sopra  
   const volatile Pappo &cv\_pappetto = pappetto;// OK! più qualificato  
   // Peppo &r\_peppo = pappo; // ERRORE! pappo non c'entra con Peppo  
   Ciccio &r\_ciccio\_da\_peppo = (Ciccio &)peppo;//....  
   // ...... ACCETTATO! fornita conversione  
   Peppo && rif\_destro = Peppo(  ); // vita di Peppo(  ) prolungata  
   // Peppo & rif\_sinistro = Peppo(  ); // ERRORE! riferimento sinistro non const  
   const Peppo & rif\_sinistro\_ok = Peppo(  ); // QUESTO SÌ  
   Peppo &r\_peppo = peppo;  
   // Peppo &&rd\_peppo = peppo; // ERRORE! rif. destro da lvalue  
   }*
7. non si ritengono indispensabili esempi per questa forma di inizializzazione, del resto già incontrata in precedenza;
8. ...e neppure di questa
9. ... o di questa
10. ... o di questa
11. ... o di quest'altra
12. e, per finire, un esempio che illustra la *brace\_or\_equal*:  
      
    *# include <iostream>  
    using namespace std;  
      
    struct Ciccio  
    {  
    int k; // inizializzazione default  
    int n = 1; // brace\_or\_equal  
    int m{1}; // brace\_or\_equal  
    int i{2}; // brace\_or\_equal, ignorata dal costruttore default...  
    Ciccio(  ) : i(3) {  } //... ma onorata da quest'altro:   
    Ciccio(int s) : k(s) {  }  
    };  
      
    int main(  )  
    {  
    Ciccio c, d(90);  
    cout << c.i << ' ' << c.m << ' ' << c.n << ' ' << c.k << '\n';  
    cout << d.i << ' ' << d.m << ' ' << d.n << ' ' << d.k << '\n';  
    }*

**Canto trentacinquesimo: tutta la verità sui costruttori**

Attraverso questo percorso, e anche in tutto il resto della presente documentazione, in quasi ogni esempio in cui si fa riferimento a classi si sono fatte ampie citazioni delle funzioni costruttore, o distruttore, della classe medesima.  
In questo canto l'argomento viene trattato con maggior attenzione e sistematicità e più approfonditamente, fino a un livello esauriente per le vostre esigenze.   
  
Come ormai dovrebbe essere noto, i costruttori di una classe sono delle funzioni membri con lo stesso nome della classe, e che si distinguono quindi tra loro SOLO per le diverse liste di argomenti ricevuti: i costruttori sono pertanto l'esempio più utile, perché quasi irrinunciabile, dell'applicazione della regola dell'*overload* delle funzioni. Per la loro stessa natura i costruttori sono anche le sole funzioni della galassia per cui non occorre dichiarare il tipo restituito, semplicemente perché tipo e nome coincidono. Per la salvaguardia della pignoleria, anche le funzioni *operator* per il *casting* hanno nome e tipo restituito collassati insieme, ma in quel caso il nome può essere, spesso, anche una parola di vocabolario.   
  
Una classe, appena comincia a essere definita al compilatore, ossia appena si scrive la graffa aperta che ne inizia la descrizione, viene da questi gratificata di TRE costruttori PUBBLICI, UN distruttore PUBBLICO e DUE funzioni *operator* PUBBLICHE in *overload* per l'operatore di assegnamento che abbia come operandi DUE istanze della classe, salvo eccezioni che saranno indicate appresso, in questo stesso documento.  
  
Questi SEI metodi membri ci sono con efficacia, almeno fino a quando il programmatore non assume in proprio la responsabilità di inserirli personalmente ed esplicitamente nella definizione della propria classe.  
Le loro dichiarazioni sottintese sono le seguenti, detto *C* il nome della classe che, ovviamente, è arbitrio del programmatore:

* *C(  );* // costruttore di *default*, d'ora in avanti *dctor*
* *~C(  );* // distruttore, d'ora in avanti *distr*
* *C(const C &);* // costruttore di copia (*copy constructor*), da ora *cctor*
* *C(C &&);* // costruttore di trasferimento (*move constructor*), da ora *mctor*
* *C & operator = (const C &);*   
  // operatore di assegnamento con copia  
  // (*copy assignment operator*), da ora *copeq*
* *C & operator = (C &&);*   
  // operatore di assegnamento con trasferimento  
  // (*move assignment operator*), da ora *mopeq*

Questi SEI metodi membri sono detti triviali se:

* *dctor* non fa assolutamente nulla, eccetto, ovviamente, prendere la memoria necessaria all'oggetto da costruire;
* *distr* non fa assolutamente nulla, eccetto, ovviamente, rilasciare la memoria occupata dall'oggetto da distruggere;
* *cctor* e *mctor* fanno esattamente la stessa cosa, ossia prendere la memoria necessaria per la costruzione della copia dell'oggetto ricevuto come argomento, realizzandovi dentro la STESSA SEQUENZA BINARIA di *bit* accesi e spenti dell'oggetto originale;
* *copeq* e *mopeq* fanno anch'essi esattamente la stessa cosa, ossia riprodurre nell'operando di sinistra la STESSA SEQUENZA BINARIA di *bit* accesi e spenti dell'operando di destra.

**NON PER TUTTE** le classi questi metodi possono rispondere alle predette caratteristiche ed essere quindi denominati triviali: affinché possa accadere occorre che la classe cui appartengono NON SIA una classe polimorfa, NON ABBIA antenate da cui eredita in modo *virtual* e NON ABBIA variabili membro NON qualificate *static* che siano a loro volta istanze di altre classi SPROVVISTE di omologhi metodi qualificabili come triviali.  
  
Pertanto, finché uno si limita a definire una classe *C* che non sia erede di nessuno, non abbia eredi a sua volta e abbia membri che appartengono solo a tipi nativi, È CERTO che una simile classe ha tutti questi sei metodi qualificabili come triviali ed è per questa ragione che funziona perfettamente anche se il programmatore non spreca una riga di codice per definire qualcuno di questi sei metodi.  
  
Appena però si comincia a essere un po' più evoluti, e si comincia a fare cose appena un po' più complicate, cominciano le eccezioni che impediscono al compilatore di equipaggiare silenziosamente una classe con TUTTI i metodi in questione, di modo che qualcuno può cominciare a venir meno.

* *dctor* NON VIENE PIÙ FORNITO DAL COMPILATORE appena il programmatore inserisce nella classe, di propria iniziativa, un suo costruttore qualsiasi, anche con una lista di argomenti diversa da *dctor*; in tal caso *dctor* può essere ricuperato in due modi: o definendolo esplicitamente, in aggiunta all'altro costruttore, oppure limitandosi a dichiararlo in questo modo, sempre usando *C* come nome della classe:   
  *C(  ) = default;*
* Il *dctor* fornito dal compilatore è addirittura cancellato, e quindi reso irrecuperabile e definitivamente inutilizzabile per una classe *C* per cui si verifichi anche uno solo dei fatti seguenti:
  1. un membro della classe è un riferimento, ed è privo di inizializzazione *brace\_or\_equal*: il compilatore non può applicare a un membro siffatto l'inizializzazione *default*.
  2. un membro della classe è qualificato *const* ed è privo di inizializzazione *brace\_or\_equal*, OVVERO, se il suo tipo è una classe a sua volta, la SUA classe è priva di un *dctor* fornito dal programmatore (ricordate? RICORDATE!)
  3. un membro della classe ha un *dctor* che è stato anche lui cancellato, per una qualsiasi delle ragioni che si stanno or ora discutendo, OVVERO per decisione del programmatore stesso che, in QUELLA CLASSE (sia D il di lei nome) l'ha dichiarato così:  
     *D(  ) = delete;*
  4. la classe ha un'antenata diretta o *virtual* il cui *dctor* ha subìto la sorte di cui al punto precedente;
  5. la classe ha un'antenata diretta o *virtual* il cui distruttore ha subìto la sorte di cui al punto precedente;
  6. la classe è una *union* un cui membro ha un *dctor* NON triviale;
  7. la classe è una *union* e TUTTI i suoi membri sono qualificati *const*.
* Il *distr* fornito dal compilatore è cancellato, nel senso sopra spiegato, se per la sua classe si verifica anche uno solo dei seguenti fatti:
  1. la classe ha un membro NON *static* il cui *distr* è cancellato a sua volta (vedi punto 3. dell'elenco precedente questo);
  2. la classe ha un'antenata diretta o *virtual* il cui *distr* è cancellato a sua volta (vedi punto 4. dell'elenco precedente questo);
  3. la classe è una *union* con un membro che ha un *distr* non triviale (vedi punto 6. dell'elenco precedente questo);
  4. la classe appartiene a una gerarchia il cui capostipite ha un *distr* qualificato *virtual* e non è fornito per la classe un valido *overload* per l'operatore *delete*.

L'uso di distruttori qualificati *virtual* in una classe capostipite di una gerarchia polimorfa è raccomandabile proprio in virtù di quest'ultimo punto: in tal modo il programmatore si autoimpone, se non vuol correre rischi di incorrere in *undefined behaviour*, O di fornire un proprio *operator delete* O di NON invocarlo esplicitamente.  
Qui si informa ogni lettore che si incorre in *undefined behaviour* se si invoca l'operatore *delete* su un puntatore a una classe capostipite inizializzato con un puntatore a una classe erede E NON SI È OTTEMPERATO A QUEST'ULTIMO PUNTO DELL'ELENCO.

* Il *cctor* fornito dal compilatore ha la dichiarazione sottintesa citata sopra se ce l'hanno così TUTTI i *cctor* di ogni membro NON *static* e TUTTI quelli di ogni eventuale antenata diretta o *virtual*; altrimenti la dichiarazione sottintesa perde la qualifica *const* nell'argomento. È consentito a una classe avere entrambi i *cctor* in *overload*.  
  Il *cctor* NON VIENE FORNITO DAL COMPILATORE se il programmatore ne fornisce uno proprio, ma si può indurre il compilatore a fornirlo ugualmente anche solo ricorrendo alla dichiarazione terminata con la clausola *= default;* (vedi sopra).  
  Tuttavia il *cctor* fornito dal compilatore è cancellato, col solito significato, se per la sua classe si verifica anche uno solo dei seguenti fatti:
  1. in analogia con gli elenchi precedenti, la classe ha un membro NON *static* o una classe antenata diretta o *virtual* col *cctor* cancellato o comunque inaccessibile;
  2. la classe ha un'antenata diretta o *virtual* col *distr* cancellato o comunque inaccessibile;
  3. in analogia con gli elenchi precedenti, la classe è una *union* con un membro dotato di un *cctor* non triviale;
  4. la classe ha un membro che è un riferimento **destro**;
  5. il programmatore ha fornito o un suo *mctor* o un suo *mopeq*.

Va osservato che, in base a regole di ottimizzazione ampiamente consolidate, l'esecuzione del *cctor* può essere omessa in numerose situazioni in cui potrebbe anche essere attesa: ne segue che un *cctor* fornito da un buon programmatore non dovrebbe MAI produrre effetti collaterali tali da condizionare il funzionamento del programma secondo se il *cctor* sia eseguito oppure no.

* Il *mctor* fornito dal compilatore viene cancellato, nella solita accezione del termine, se per la classe cui apparterrebbe si verifica uno qualsiasi dei seguenti fatti:
  1. con le ormai consuete analogie, la classe ha un membro NON *static* o una classe antenata diretta o *virtual* col *mctor* cancellato o comunque inaccessibile;
  2. la classe ha un'antenata diretta o *virtual* col *distr* cancellato o comunque inaccessibile;
  3. la classe è una *union* con almeno un membro dotato di un *cctor* (ho scritto *cctor*) non triviale;
  4. la classe ha un'antenata diretta o *virtual* PRIVA di un *mctor* e ANCHE di un *cctor* TRIVIALE.

Il *mctor*, anche se non cancellato, e quindi ricuperabile con la dichiarazione terminata con la clausola *= default;*, NON VIENE COMUNQUE FORNITO PIÙ dal compilatore in numerosi casi, ossia se si verifica una qualsiasi delle seguenti circostanze:

* 1. il programmatore ha fornito esplicitamente un suo *mctor*; in questo caso può essere ricuperato quello fornito dal compilatore dichiarandolo in legittimo *overload* col proprio e finendo per avere DUE *mctor* con le dichiarazioni concorrenti, e accettate,  
     *C(C&&);*  
     *C(const C&&);*
  2. il programmatore ha fornito esplicitamente un suo *cctor*: anche in questo caso, e purché il *mctor* implicito non sia stato cancellato, lo si può recuperare come già detto;
  3. il programmatore ha fornito esplicitamente un suo *copeq*, con eventuale recupero come sopra;
  4. il programmatore ha fornito esplicitamente un suo *mopeq*, con eventuale recupero come sopra;
  5. il programmatore ha fornito esplicitamente un suo *distr*, con eventuale recupero come sopra;
* Anche il *copeq* può esistere con diversi *overload* nella classe campione *C*: tralasciando gli infiniti *overload* possibili quando l'operando destro ha un tipo diverso da *C*, i quali non concernono i discorsi che si stanno tenendo, possono darsi le seguenti dichiarazioni, dalle quali si omette l'esplicitazione del tipo restituito perché anche quello può essere qualsiasi, quando a scrivere la funzione è il programmatore:
  1. *tipo\_restituito operator=(const C &);*
  2. *tipo\_restituito operator=(C &);*
  3. *tipo\_restituito operator=(C);*

Ovviamente la seconda e la terza di queste segnature di *copeq* NON POSSONO COESISTERE.  
Il *copeq* fornito dal compilatore, come anticipato, ha la prima delle tre segnature e *tipo\_restituito* è *C&*, a meno che qualche variabile membro NON *static* o qualche classe antenata diretta abbia un *copeq* con una segnatura più debole, ossia una delle altre due: in tal caso il *copeq* fornito dal compilatore assume la seconda segnatura, sempre mantenendo fermo il tipo restituito *C&*.  
Il *copeq* fornito dal compilatore viene comunque cancellato se si verifica almeno uno dei seguenti fatti:

* 1. la classe ha una variabile membro NON *static* qualificata *const*: il compilatore non si assume la responsabilità di porla come membro dell'operando sinistro;
  2. la classe ha una variabile membro NON *static* che sia un riferimento: idem come sopra;
  3. come al solito, la classe ha una variabile membro NON *static* o un'antenata diretta o *virtual* che abbia il *copeq* cancellato o comunque inaccessibile;
  4. il programmatore ha fornito esplicitamente un suo *mctor*;
  5. il programmatore ha fornito esplicitamente un suo *mopeq*;
* per quanto riguarda il *mopeq* si possono ripetere pedissequamente i ragionamenti fatti per il *copeq*, con la differenza che stavolta le possibili segnature sono solo due, ossia con l'argomento che è *C&&* oppure *const C&&*, con la prima delle due segnature preferita dal compilatore, e con la possibilità della coesistenza di entrambe, anche quando una delle due fosse fornita dal programmatore, sempre attraverso la clausola finale *= default;* posta nella dichiarazione della funzione che si vuol ricuperare.  
  Anche le condizioni per cui il *mopeq* fornito dal compilatore risulta cancellato, e quindi non ricuperabile, sono le stesse, *mutatis mutandis* date per il *copeq* con l'aggiunta dell'eventualità che la classe abbia la solita variabile membro o la solita antenata PRIVA di un *mopeq* e che neppure abbia un *cctor* TRIVIALE.  
  Infine il compilatore si astiene dal fornire il *mopeq*, pur non cancellandolo, quando il programmatore fornisce esplicitamente anche solo uno fra *cctor*, *mctor*, *copeq* o *distr*.

Il seguente programma, che riprende e amplia un codicillo ideato dallo studente Davide del Moro (2015) con l'evidente e lodevole intento di capirci qualcosa, illustra appunto il quando, il come e il perché sono eseguite le funzioni membro qui a lungo dibattute: per non farsi mancare nulla vi sono inseriti perfino DUE distinti *overload* di ciascuno degli *operator=*.   
  
*# include <iostream>  
using namespace std;  
  
struct C  
{  
C(  ) {clog << "sono il default ctor\n" << flush;}  
~C(  ) {clog << "sono il distruttore\n" << flush;}  
/\*explicit \*/C(const C& c) {clog << "sono il copy ctor\n" << flush;}  
C(C &&c) {clog << "sono il move ctor\n" << flush;}  
C &operator=(C &c) {clog << "sono il copy =\n" << flush; return c;}  
C &operator=(const C &c)  
{clog << "sono il copy const=\n" << flush; return const\_cast<C&>(c);}  
C &&operator=(C &&c)  
{clog << "sono il move =\n" << flush; return move(c);}  
C &operator=(const C&& c)  
{clog << "sono il move const =\n" << flush; return const\_cast<C&>(c);}  
C operator~(  ) {clog << "sono l'operator~\n"; return \*this;}  
};  
  
int main(  )  
{  
C c;  
clog << "111111111111111111111111\n" << flush;  
C d(c);  
clog << "222222222222222222222222\n" << flush;  
C e(move(d));  
clog << "333333333333333333333333\n" << flush;  
C f(~e);  
clog << "444444444444444444444444\n" << flush;  
c = ~f;  
clog << "555555555555555555555555\n" << flush;  
f = c;  
clog << "666666666666666666666666\n" << flush;  
C g = f;  
clog << "777777777777777777777777\n" << flush;  
C h = move(g);  
clog << "888888888888888888888888\n" << flush;  
const C i;  
h = i;  
clog << "999999999999999999999999\n" << flush;  
h = move(i);  
}*   
  
Va notato il commento che esclude la qualifica *explicit* dalla dichiarazione del costruttore di copia; se la si ripristinasse sarebbero segnalati ben DUE errori: il primo nella compilazione della funzione *operator~*, che non saprebbe più come restituire l'oggetto *\*this*, e il secondo all'atto della dichiarazione dell'oggetto *g* in *main*, che non potrebbe più essere costruito partendo dall'oggetto inizializzante *f*.  
  
Esaurito l'argomento delle funzioni membro eventualmente regalate o sottaciute o abolite dal compilatore, al programmatore resta facoltà di aggiungere alla classe *C* QUANTI DIVERSI COSTRUTTORI desidera e anche, come detto, quanti *copy/move assignment operator* voglia, ovviamente con operando destro di tipi diversi; non potrà però MAI corredare la classe con più di UN SOLO DISTRUTTORE, quello di cui si è parlato finora. Si potrebbe introdurre, a tal proposito, questo nuovo assioma:  
  
**di distruttori ce n'è UNO, di costruttori anche trentuno**.   
  
Dovrebbe apparire abbastanza superfluo dire che ALMENO UNO dei costruttori debba essere pubblico e del resto SEMPRE pubblici sono i TRE di cui si è parlato; non è tuttavia affatto vietato che vi siano costruttori NON pubblici, coi quali, ad esempio, la classe possa istanziare come meglio le giova dei propri oggetti dall'interno dei suoi stessi metodi: di tali costruttori l'utilizzatore ultimo del programma che non ne sia anche l'autore potrebbe perfino ignorare del tutto l'esistenza.  
  
E non è affatto inconsueto inserire APPOSTA un *cctor*, ad esempio, nella zona NON PUBBLICA della classe, proprio per IMPEDIRE che oggetti di tale classe possano essere copiati, vale a dire per INIBIRE totalmente l'inizializzazione per copia di tali oggetti, e, per quanto detto sopra, inibire anche la copia di qualsiasi oggetto che li avesse come propri membri.  
  
Quello che è meno scontato è che si capisca QUANTA ESECUZIONE DI CODICE possa nascondersi dietro una banale dichiarazione come questa:  
  
{Ciccio ciccio;}   
  
in cui un oggetto *ciccio*, istanza di una certa classe *Ciccio*, viene dichiarato e inizializzato con inizializzazione *default* (lo si sa, ora, vero?), entro un certo ambito che, lo si noti, NON CONTIENE ALTRO. Le tavole della legge del linguaggio ci assicurano che *ciccio* NON ESISTE prima che si apra la graffa e NON ESISTE PIÙ dopo che la graffa si è chiusa. Nel frattempo, tra una graffa e l'altra, che cosa succede?  
  
Evidentemente *ciccio* deve essere chiamato all'esistenza e poiché subisce l'inizializzazione *default* è NECESSARIO che la classe *Ciccio* sia equipaggiata con un *dctor*, implicito, esplicito, triviale o no che sia. Sarà lui che, eseguendosi, lo costruirà; e già qui si vede che la nostra banale dichiarazione comporta senz'altro ALMENO l'esecuzione del codice contenuto nel costruttore di *default* di *Ciccio*.  
  
Ma non è tutto: *Ciccio* potrebbe essere l'epigona di una lunga discendenza di classi, ad esempio di tante classi quante sono le lettere dell'alfabeto (e perché NO?) in quell'ordine.  
Allora, per istanziare *ciccio*, occorre istanziare ANCHE TUTTE LE CLASSI da cui *Ciccio* dipende, nel dovuto ordine: pertanto per prima si deve istanziare la classe *A*, poi la classe *B*...fino alla classe *Z*, da cui *Ciccio* discende direttamente: in totale la nostra dichiarazione comporta l'esecuzione ordinata di ben 27 (ventisette) costruttori (e ci siamo fermati a ventisette...per non arrivare a 27000).  
  
Non è detto che TUTTI i costruttori eseguiti siano quelli di *default* delle varie classi, anche se POTREBBE ESSERE, perché, come dovremmo sapere, NULLA IMPEDISCE al costruttore *default* della classe *Z* (ad esempio) di invocare dalla propria lista di inizializzazione un costruttore NON *default* di *Y* ... e così via ... aprendo un numero di possibilità diverse che cresce esponenzialmente col numero di livelli e col numero di costruttori definiti in ciascun livello...  
  
Ma non è tutto: ognuna delle nostre classi ha, per natura, dei membri. E nulla vieta che i membri appartengano ad altre classi, anche estranee alla gerarchia (perché NO?); pertanto, quando si istanzia la classe *K*, BISOGNA ISTANZIARE ANCHE TUTTI I SUOI MEMBRI e se UNO (UNO ?) di tali membri appartiene alla classe *Pappo* bisogna invocare ed eseguire un costruttore di *Pappo*, la quale potrebbe (perché NO?) essere l'epigona di una diversa gerarchia... allora per istanziare *Pappo* occorre PRIMA istanziare, nell'ordine dovuto, TUTTE LE SUE ANTENATE ... ... e così via ... NON ALL'INFINITO, **OVVIAMENTE**.  
  
Per quanto sia vero che l'algoritmo sia FINITO...penso che si sia potuto comprendere che dietro quell'apparentemente banale dichiarazione, che potrebbe essere un'intera funzione *main*, è potenzialmente nascosta TUTTA LA FISICA dal *big bang* a oggi.  
  
Quando la parentesi graffa si chiude, e quindi l'oggetto *ciccio*, tanto faticosamente costruito, perde la propria vita, TUTTA l'opera di costruzione appena conclusa al punto e virgola si riconverte in un'opera di distruzione: sono eseguiti, nell'ordine INVERSO, TUTTI I DISTRUTTORI disponibili di ogni classe coinvolta, di modo che si torna al nulla precedente ogni creazione ... salvo eventuale radiazione di fondo costituita da eventuali membri *static* di qualche classe, che scompaiono solo se la graffa che si chiude è proprio quella che chiude *main* (BIBLICO LINGUAGGIO...).  
  
Il seguente programma che, lungi dal racchiudere in sé tutto lo scibile umano, è del tutto inutile, ha però il merito di far toccare con mano quanto è stato appena detto; eseguitelo pertanto con deferenza.

*# include <iostream>  
using namespace std;  
  
class Pappo  
{  
public:  
~Pappo(  )  
{cout << "distruttore di Pappo\n";}  
Pappo(  )  
{cout << "costruttore default di Pappo\n";}  
};  
  
class Peppo : public Pappo  
{  
public:  
~Peppo(  )  
{cout << "distruttore di Peppo\n";}  
Peppo(  )  
{cout << "costruttore default di Peppo\n";}  
};  
  
class Pippo : public Peppo  
{  
public:  
~Pippo(  )  
{cout << "distruttore di Pippo\n";}  
Pippo(  )  
{cout << "costruttore default di Pippo\n";}  
};  
  
class A  
{public:  
Pippo pippo;  
~A(  )  
{cout << "distruttore di A\n";}  
A(  )  
{cout << "costruttore default di A\n";}  
A(int x)  
{cout << "costruttore(int) di A\n";}};  
  
class B : private A  
{public:  
Peppo peppo;  
~B(  )  
{cout << "distruttore di B\n";}  
B(  )  
{cout << "costruttore default di B\n";}  
B(int x)  
{cout << "costruttore(int) di B\n";}  
B(int x, int y) : A(y)  
{cout << "costruttore(int, int) di B\n";}};  
  
class Z : protected B  
{public:  
Pappo pappo;  
~Z(  )  
{cout << "\n\n\ndistruttore di Z\n";}  
Z(  )  
{cout << "costruttore default di Z\n\n\n";}  
Z(int x)  
{cout << "costruttore(int) di Z\n\n\n";}  
Z(int x, int y) : B(y)  
{cout << "costruttore(int, int) di Z\n\n\n";}  
Z(int x, int y, int z) : B(y, z)  
{cout << "costruttore(int, int, int) di Z\n\n\n";}  
Z(int x, int y, int z, int w)  
{cout << "costruttore(int, int, int, int) di Z\n\n\n";}};  
  
int main(  )  
{Z z0, z1(1), z2(2, 3), z3(4, 5, 6), z4(7, 8, 9, 0);}*

Come qualsiasi altra funzione membro, anche ogni costruttore e il distruttore possono essere SOLO dichiarati nell'ambito della classe di pertinenza e definiti all'esterno, per renderla COMPLETA. Le sintassi sono ovvie e, sempre utilizzando *C* come nome della classe, sono:  
  
*C :: C (  ) /\*omissis\*/ {/\*omissis\*/} // definizione esterna di dctor  
C :: ~C(  ) {/\*omissis\*/} // definizione esterna di distr*Un costruttore può subappaltare in tutto o in parte l'istanziazione dell'oggetto che è chiamato a costruire a un (e un solo) altro costruttore della sua classe, ANCHE NON PUBBLICO, con la stessa sintassi utilizzata per invocare costruttori di classi antenate. Il cosiddetto costruttore delegato viene eseguito, coerentemente, prima di quello delegante, il quale, però, non può poi inserire nella propria lista di inizializzazione altro che la delega. A titolo di esempio autoesplicativo si provi a eseguire il breve codice che segue:  
  
*# include <iostream>  
using namespace std;  
  
struct Ciccio  
{  
Ciccio(  )  
{cout << "sono il costruttore default\n";}  
Ciccio(int u)  
{cout << "sono Ciccio(int u) con u = "<< u << endl;}  
Ciccio(double u) : Ciccio(12)  
{cout << "sono Ciccio(double u) con u = " << u << endl;}  
Ciccio(float u) : Ciccio("trallallero")  
{cout << "sono Ciccio(float u) con u = " << u << endl;}  
private:  
Ciccio(const char \* p)  
{cout << "sono Ciccio(const char \*p) con p = " << p << endl;}  
};  
  
int main(  )  
{cout << "istanzio un Ciccio default\n"; Ciccio ciccio;  
cout << "ora istanzio un Ciccio(int)\n"; Ciccio caccio(1);  
cout << "ora istanzio un Ciccio(double)\n"; Ciccio coccio(2.1);  
cout << "ora istanzio un Ciccio(float)\n"; Ciccio cuccio(4.0f);  
}*Non dovrebbe esservi chi non si avveda che, in questo modo, nel caso di classi complicate, dotate di numerosi costruttori, si evita la duplicazione inutile di un grande numero di linee di codice. Si osservi che il costruttore *Ciccio(int)* è utilizzato sia come costruttore unico, nell'istanziazione di *caccio*, sia come costruttore delegato nell'istanziazione di *coccio*, mentre il costruttore *Ciccio(const char \*)*, essendo PRIVATO, può SOLO concorrere all'istanziazione di *cuccio* in veste di costruttore delegato da *Ciccio(float)*.   
  
In una gerarchia ereditaria di classi ciascuna di esse può ereditare i costruttori della sua antenata diretta che non siano *dctor*, *cctor* o *mctor* i quali, come si sa, seguono regole proprie, evitando, anche in questo modo, al programmatore di dover duplicare inutilmente molte linee di codice, dato che l'alternativa sarebbe quella di riscrivere nella classe figlia un costruttore con la stessa lista di argomenti di uno della classe madre, al solo scopo di trasferire a quest'ultimo gli argomenti ricevuti invocandolo dalla lista di inizializzazione.   
Ciò si ottiene usando nella definizione della classe erede una linea che reciti  
  
*using Mamma::Mamma;*  
  
ove *Mamma* è, ovviamente, il nome della classe madre. Nulla impedisce di inserire comunque nella classe erede dei costruttori di ugual segnatura rispetto a quelli ereditati, i quali, in tal caso, risulteranno coperti da quelli esplicitamente specificati per la classe erede. A maggior ragione la classe erede potrà essere equipaggiata con tutti i costruttori che si desiderano e che abbiano liste di argomenti del tutto diverse da quelli ereditati. Ecco alcuni brevi esempi.

**Esempio 1:**  
*# include <iostream>  
using namespace std;  
  
struct Nonna  
{  
Nonna(  )  
{cout << "sono Nonna(  )\n";}  
Nonna(int a)  
{cout << "sono Nonna(int)\n";}  
};  
  
struct Mamma : Nonna  
{  
using Nonna :: Nonna;  
/\*  
questa classe ha un dctor, un cctor, un mctor e  
un costruttore Mamma(int) equivalente a  
Mamma(int k) : Nonna(k) {  }  
\*/  
};  
  
struct Figlia : Mamma  
{  
using Mamma :: Mamma;  
/\*  
questa classe ha un dctor, un cctor, un mctor e  
un costruttore Figlia(int) equivalente a  
Figlia(int k) : Mamma(k) {  }  
\*/  
};  
  
int main(  )  
{  
/\*  
per quanto riportato nei precedenti commenti  
le due seguenti istanziazioni sono del tutto lecite.  
\*/  
Figlia figlia1, figlia2(0);  
}***Esempio 2:**  
*# include <iostream>  
using namespace std;  
  
struct Nonna  
{  
Nonna(  )  
{cout << "sono Nonna(  )\n";}  
Nonna(int a)  
{cout << "sono Nonna(int)\n";}  
};  
  
struct Mamma : Nonna  
{  
using Nonna :: Nonna;  
/\*  
questa classe ha un dctor, un cctor, un mctor e  
un costruttore Mamma(int) equivalente a  
Mamma(int k) : Nonna(k) {  }  
\*/  
};  
  
struct Figlia : Mamma  
{  
using Mamma :: Mamma;  
Figlia(int k)  
{cout << "sono Figlia(int)\n";}  
/\*  
questa classe "copre", con un proprio  
costruttore, quello di ugual dichiarazione ereditato  
da Mamma; ma per la stessa ragione* ***perde*** *il proprio dctor. \*/  
};  
  
int main(  )  
{  
/\*  
per quanto riportato nei precedenti commenti  
solo la seconda istanziazione è lecita.  
\*/  
Figlia // figlia1, // ERRORE: Figlia NON HA dctor  
figlia2(0);  
}***Esempio 3:**  
*# include <iostream>  
using namespace std;  
  
struct Nonna  
{  
Nonna(  )  
{cout << "sono Nonna(  )\n";}  
Nonna(int a)  
{cout << "sono Nonna(int)\n";}  
};  
  
struct Mamma : Nonna  
{  
using Nonna :: Nonna;  
Mamma(  )   
{cout << "sono il DOVUTO dctor di Mamma\n";}  
Mamma(double x, int k=0, char c='a', float p=1.4f)  
{cout <<   
"sono Mamma(double, int=, char=, float=)\n";}  
/\*  
questa classe ha un dctor (definito per necessità),  
un cctor, un mctor, il costruttore esplicitamente  
definito sopra e un costruttore Mamma(int)   
ereditato da Nonna ed equivalente a  
Mamma(int k) : Nonna(k) {  }  
\*/  
};  
  
struct Figlia : Mamma  
{  
using Mamma :: Mamma;  
/\*  
questa classe ha un dctor, un cctor, un mctor ed  
eredita da Mamma  
TUTTI I SEGUENTI COSTRUTTORI:  
Figlia(int)  
Figlia(double)  
Figlia(double, int)  
Figlia(double, int, char)  
Figlia(double, int, char, float)  
senza che ne sia stato scritto neppure uno.  
\*/  
};  
  
int main(  )  
{  
/\*  
per quanto riportato nei precedenti commenti  
tutte le seguenti istanziazioni sono lecite.  
\*/  
Figlia  
figlia1,  
figlia2(0),  
figlia3(0.0),  
figlia4(1.0, 1),  
figlia5(2.4, 3, 'z'),  
figlia6(3.1, 8, 0, 0);  
}*

L'ultimo esempio mostra come un costruttore provvisto di argomenti *standard* venga ereditato in tutte le sue possibili accezioni, vale a dire che gli argomenti *standard* NON SONO EREDITATI.

**Canto trentaseiesimo: tutta la verità su *lvalues*, *rvalues*, riferimenti sinistri e destri e *"simile lordura"*; in due parole: la *move semantic***

Questo canto va letto presupponendo che ci si siano fatte le ossa su tutto il resto, fino a un punto tale da non avere più gravi dubbi sulle pieghe fini del linguaggio. In effetti gli argomenti che saranno qui introdotti sono più fini delle stesse citate pieghe e per giunta, se non si hanno particolari e pressanti esigenze di ottimizzazione, possono anche restare sconosciuti a un programmatore apprendista.  
  
Nondimeno, la stessa finezza appena adombrata potrebbe e dovrebbe indurre alla lettura ATTENTA quelli che si ritengano non nati *"a viver come bruti"*, ma piuttosto portati a *"seguir virtute e canoscenza"*.  
  
Fatte queste premesse, e venendo al sodo, con le parole *move semantic* si intende un insieme di regole nuove, introdotte dallo *standard* 2011, che alterano i criteri per la realizzazione delle copie di variabili e di oggetti, tendendo a minimizzarle e/o a eliminarle del tutto ogni volta che ciò sia possibile senza pregiudicare il buon esito di un programma, anzi, al contrario, ottenendone un significativo miglioramento delle prestazioni.  
  
Prima di addentrarsi nell'esposizione di tali regole è opportuna una breve rassegna sulle cosiddette categorie di valori cui può essere fatta appartenere una data espressione; prima dell'avvento del C++, ossia già al tempo del C ANSI, di tali categorie se ne conoscevano DUE, chiamate, con biasimevole termine anglofono, *lvalues* e *rvalues* e si distinguevano una dall'altra essenzialmente per il fatto che le espressioni *lvalue* potevano trovarsi a sinistra (*l* sta appunto per *left*) dell'operatore di assegnamento e le altre no (*r* sta ovviamente per *right*).  
Nelle due seguenti linee di codice appare evidente quanto detto:  
  
*i = x + y; // lvalue = rvalue  
\*(z + 1) = 3; // lvalue = rvalue*Con l'avvento del C++, ma prima dello standard 2011, le categorie rimasero le stesse, ma venne data una definizione più ampia di *lvalue*, per potervi comprendere espressioni che NON ESISTEVANO nel C ANSI, e si disse che *lvalue* era qualsiasi espressione di cui fosse possibile prendere l'indirizzo in memoria. In questo modo diventava *lvalue* anche l'invocazione di una funzione che restituisse appunto un riferimento, fatto impossibile nel C ANSI, e che non per caso finiva col potersi scrivere anch'essa a sinistra del segno =. Quanto a *rvalue* si disse semplicemente che tutto ciò che non è *lvalue* è *rvalue*.   
  
Nel nuovo standard 2011 sono introdotte ben CINQUE categorie, aggiungendo alle classiche *lvalues* e *rvalues* le nuove *xvalues*, *glvalues* e *prvalues*. Come si può intuire il linguaggio si fa assai più puntiglioso e pignolo, sotto questo punto di vista, ma la buona notizia è che, a un livello iniziale di apprendimento, si può tranquillamente prescindere dalle tre nuove categorie.  
  
Siccome però qui si suppone di non essere a un livello iniziale di apprendimento ecco qui di seguito le definizioni delle diverse categorie:

* *lvalue* è qualsiasi espressione che abbia una locazione persistente in memoria di cui possa essere ottenuto l'indirizzo attraverso l'operatore &;  
  un identificatore dichiarato, un puntatore, una funzione che restituisca un riferimento sono tutti esempi di *lvalues*. Si noti che in questa definizione rientrano gli identificatori qualificati *const*, perché se ne può ottenere l'indirizzo, quantunque NON possano comparire a sinistra dell'operatore di assegnamento, per cui, secondo il C ANSI, non sarebbero stati classificabili come *lvalues*.  
  Qualsiasi *lvalue* può comparire ovunque possa apparire un *rvalue*, definito appresso.
* *rvalue* è qualsiasi espressione di cui NON si possa ottenere l'indirizzo in memoria, essenzialmente perché tale espressione si trova in una locazione NON persistente o NON nominata; *x+1-a* è un *rvalue*, perché la locazione di memoria in cui si trova il suo valore non è né nominata né persistente; l'invocazione di una funzione che restituisca una variabile o un oggetto è un *rvalue* perché la locazione di memoria in cui l'oggetto viene restituito non è persistente. Qualsiasi *rvalue* non può trovarsi là dove sarebbe richiesto un *lvalue*.
* *xvalue* è la categoria appositamente introdotta nel nuovo *standard* in vista dell'implementazione della *move semantic* e quindi sarà la protagonista del presente documento. Volendo anticiparne una definizione già comprensibile a questo livello si potrebbe dire che è un *rvalue* che si trasforma per pochi fuggevoli istanti, prima di morire, in un *lvalue*: una specie di filugello.
* *glvalue* è il nome collettivo che mette insieme *lvalues* e *xvalues*; in effetti la *g* iniziale sta per *generalized*. Questa categoria non serve ad altro che a dire che ovunque possa trovarsi tanto un *lvalue* VERO quanto un *xvalue* spirante, DI FATTO sta un *glvalue*.
* *prvalues* sono semplicemente *rvalues* che restano pervicacemente tali per tutta la loro breve vita; in altre parole NON SONO né saranno mai *xvalues* e, a maggior ragione, MAI *lvalues*, neppure per un nanosecondo. In effetti *p* sta per *pure*.

In definitiva, delle cinque categorie elencate solo tre sono essenziali, le ultime due essendo solo orpelli o scorciatoie verbali. Per chi ne avesse inappagata la curiosità l'iniziale di *xvalues* sta per *eXpiring*, con l'intento di sottolineare che di tale categoria di espressioni è interessante solo il canto del cigno.

**Il nucleo del problema**

Il problema cui la *move semantic* dà soluzione si evince dal seguente codice esemplificativo:  
  
*class Ciccio  
{  
// una classe molto DISPENDIOSA da istanziare  
};  
  
Ciccio funza(  )  
{  
Ciccio c;  
// omissis  
return c;  
}  
  
int main(  )  
{Ciccio x;   
// omissis  
x = funza(  );  
// omissis  
}*   
  
Se non ci fosse il commento dentro la classe *Ciccio*, che sta lì apposta, il problema nominato sarebbe un falso problema e se, pur mantenendo il commento, il programma fosse scritto così com'è, con la funzione *funza* eseguita una sola volta, ugualmente andare a fare le pulci al codice sarebbe tempo sprecato.  
  
Ma se la funzione *funza* dovesse essere eseguita, magari entro qualche ciclo nascosto negli *omissis*, per qualche decina di miliardi di volte, forse mette conto analizzare in fino che cosa succede nel programma precedente prima dell'avvento dello *standard* 2011, immaginando, ovviamente, che la logica del codice non possa essere in alcun modo cambiata, trattandosi di un esempio.   
  
La funzione *funza* istanzia un oggetto della classe *Ciccio*, nel proprio ambito, ogni volta che è invocata e, dopo averne fatto quello che le serve nei suoi *omissis*, lo restituisce per valore a *main*.  
  
Questa, a sua volta, lo riceve e lo assegna all'oggetto *x*, istanziato nel proprio ambito. Tutto ciò dovrebbe essere chiaro a ogni lettore del presente documento e, in effetti, questo codice funziona senza bisogno di aspettare il nuovo *standard*. Ma quali sono le implicazioni fini? Si possono riassumere nelle seguenti azioni:

1. Tutte le risorse di memoria impegnate per l'oggetto *c*, che è un *lvalue* nell'ambito di *funza*, devono essere trasferite (copiate) in una locazione che sia un *rvalue* nell'ambito di *main*.
2. Una volta esaurito il punto precedente, tutte le risorse di *c* devono essere rilasciate, per uscita dall'ambito di visibilità di *funza*.
3. Tutte le risorse di memoria impegnate in precedenza per l'oggetto *x*, nell'ambito di *main*, devono essere rilasciate, perché *x* è un *lvalue* che sta per essere riassegnato.
4. Le risorse di memoria impegnate per l'*rvalue* restituito da *funza* devono essere trasferite (copiate) nella locazione permanente del *lvalue x*.
5. L'*rvalue* restituito da *funza* SPIRA e le sue risorse sono rilasciate.

È davvero sorprendente quanti fatti siano nascosti dietro una sola linea di codice: non ci dovrebbe essere nessuno che non capisca quanto gravosa possa rivelarsi questa situazione se ripetuta un gran numero di volte e se si ammette che la classe *Ciccio* sia, come si è detto, assai dispendiosa.  
  
Tutti i compilatori attuali adottano ormai come criterio *standard* di ottimizzazione la cosiddetta *RVO ("return value optimization") & "copy elision"*, per cui i primi due punti del precedente elenco sono, di fatto, aboliti, sostanzialmente istanziando l'oggetto *c*, passibile di *return*, direttamente nelle stesse celle di memoria che sono etichettate come *lvalues* in *funza* e reinterpretate come *rvalues* in *main*; ma il compilatore, almeno pre-*standard* 2011, non avrebbe alcun criterio valido per evitare i successivi tre punti dell'elenco.  
  
Eppure sembra abbastanza chiara la superfluità dell'accoppiata punto 3 + punto 5: se deve essere comunque liberata della memoria, perché farlo due volte, una volta per un *lvalue* (punto 3) e un'altra per un *rvalue* destinato COMUNQUE a perire?  
  
Non sarebbe più proficuo, al punto 3, scambiare semplicemente le categorie di appartenenza e far diventare le celle di memoria di *x* non persistenti, trasformando di contro in persistenti quelle dell'*rvalue* transitorio, risparmiandosi un intero rilascio di risorse? Se si avessero due cassetti pieni e di uno dei due si dovesse buttare il contenuto, sarebbe preferibile travasare tutto da un cassetto a un altro, perché ci si ostina a voler buttare proprio QUEL cassetto, o scambiare solo le etichette dei cassetti?  
  
**In altre parole, non sarebbe meglio inventare la nuova categoria degli *xvalues*?**   
  
La risposta è chiaramente sì per ogni persona dotata di raziocinio ed è appunto ciò che lo *standard* 2011 introduce e consente.

**Quando, di fatto, si presenta il problema?**

L'esempio dato nel precedente paragrafo dovrebbe far capire che il problema si presenta essenzialmente a carico dell'operatore di assegnamento e del costruttore di copie, non tanto per quanto concerne la restituzione di oggetti (come si è visto provvede già il compilatore a questo, attraverso la *RVO*) quanto per il trasferimento per valore di argomenti a funzioni. Si potrebbe obiettare che questa seconda fattispecie potrebbe essere evitata facendo sì che le funzioni ricevano sistematicamente per riferimento i loro argomenti, ma ciò ha le seguenti forti controindicazioni, dal punto di vista sia dei programmatori sia di chi debba redigere uno standard:

1. rottura definitiva e irrevocabile della retro-compatibilità del linguaggio: le funzioni provenienti dal C ANSI NON ricevono per riferimento;
2. impossibilità di trasmettere a una funzione come argomenti degli *rvalues*;
3. necessità, al fine di neutralizzare la lacuna di cui al punto precedente, di un rimedio peggiore del male, ossia l'introduzione di un numero di *overload* di una funzione, che crescerebbe esponenzialmente col numero di argomenti, per poter tener conto di tutte le possibili combinazioni di presenza o assenza del qualificatore *const* per ciascun argomento.

Chi scrive questo documento non ritiene di dover esemplificare il punto 1; quanto ai punti 2 e 3 basta considerare il seguente breve codice:  
  
*# include <iostream>  
using namespace std;  
  
void funza(int &c)  
{cout << "sono funza(int &) ",  
cout << c << endl;}  
  
void funza(int const &c)  
{cout << "sono funza(int const &) ",  
cout << c << endl;}  
  
int main(int narg, const char \*\* args, const char \*\* env)  
{funza(narg), funza(5);}*   
  
in cui la pretesa di ricevere per riferimento ha richiesto, per un solo argomento, due *overload* di *funza* per poterne effettuare le due invocazioni compiute da *main*.  
  
Accantonata pertanto l'ipotesi di ricevere sistematicamente per riferimento gli argomenti delle funzioni e concepita l'idea della creazione degli *xvalues* come sopra definiti, ne consegue necessariamente che debba potersi accedere, almeno negli istanti conclusivi della vita di un *rvalue*, al suo indirizzo in memoria, onde poterlo far diventare, in contesti che saranno meglio precisati in seguito, un *lvalue*; nasce spontaneamente, da queste considerazioni, il concetto di *riferimento a rvalue* (in angloamericano: *rvalue reference*): un concetto autocontraddittorio con quello di *rvalue* (cfr. la definizione) e quindi, come accade sempre quando si contraddice qualcosa di consolidato, si crea qualcosa di nuovo.

**Riferimento a *rvalue***

Con questo termine lo *standard* 2011 indica appunto l'estensione del classico riferimento a *lvalue*, introdotta proprio con gli scopi che si è cercato di delineare. Come il riferimento a *lvalue* si otteneva tramite il segno grafico &, come ad esempio nei due *overload* di *funza* visti nel precedente paragrafo, così il riferimento a *rvalue* si ottiene replicando due volte lo stesso segno.  
Si consideri la seguente variante del programma presentato poco sopra:  
  
*# include <iostream>  
using namespace std;  
  
int funza(int &c)  
{cout << "sono funza(int &) ",  
cout << c << endl;  
return c;}  
  
int funza(int &&c)  
{cout << "sono funza(int &&) ",  
cout << c << endl;  
return c;}  
  
int funza(int const &c)  
{cout << "sono funza(int const &) ",  
cout << c << endl;  
return c;}  
  
int main(int narg, const char \*\* args, const char \*\* env)  
{funza(narg), funza(5), funza(funza(8));}*   
  
Rispetto alla precedente versione sono state apportate poche modifiche, la più vistosa delle quali è l'aggiunta di un terzo *overload* di *funza* che fa esattamente le stesse cose che fanno gli altri due, ma che, per la prima volta e in ossequio al nuovo *standard*, riceve appunto come argomento un riferimento a *rvalue*. Oltre a questo, tutte le *funza* sono state trasformate da *void* a *int* così da far loro restituire il proprio argomento e da consentire anche l'aggiunta dell'ultima invocazione di *funza* in *main*, ossia *funza(funza(8))*, che, ovviamente, non sarebbe stata possibile con funzioni di tipo *void*.  
Se si compila questo codice e lo si esegue senza fornire alcun argomento opzionale sulla linea di esecuzione si ottiene (provare per credere, ammesso che si abbia un compilatore aggiornato allo *standard* 2011) il seguente *output*:  
  
*sono funza(int &) 1  
sono funza(int &&) 5  
sono funza(int &&) 8  
sono funza(int &&) 8*   
  
La prima osservazione che balza all'occhio è che l'*overload* di *funza* con la segnatura *funza(int const &)* NON È PIÙ ESEGUITO. La presenza del nuovo *overload* lo ha completamente rimpiazzato, rendendolo inutile; la qual cosa, per inciso, rende priva di contenuto la precedente osservazione sul numero esponenziale di *overload* diversi che sarebbe occorsa, prima del nuovo *standard*, per poter trasmettere *rvalues* come argomenti delle funzioni quando queste li ricevono per riferimento.  
È bene ribadire che l'*overload* cortocircuitato non è stato abolito nel nuovo *standard*, tanto che il compilatore continua ad accettarlo tranquillamente; non solo, ma se si commenta il nuovo *overload*, quello cortocircuitato torna immediatamente alla ribalta nella pienezza della sua efficienza (ancora una volta, provare per credere).  
  
In tutti i casi in cui viene eseguito l'*overload funza(int &&)* esiste un *rvalue* temporaneo che, trasmesso come argomento, viene promosso a *xvalue* per poter essere trattato come *lvalue* all'interno della funzione, secondo quanto si è detto parlando delle categorie di valori in questo stesso documento.  
Si presti, però, attenzione alla seguente, ulteriore variante:  
  
*# include <iostream>  
using namespace std;  
  
int funza(int &c)  
{cout << "sono funza(int &) ",  
cout << c << endl;  
return c;}  
  
int funza(int &&c)  
{cout << "sono funza(int &&) ",  
cout << c << endl;  
return c;}  
  
int main(int narg, const char \*\* args, const char \*\* env)  
{int && c = funza(narg);  
funza(5);  
funza(funza(8));  
funza(c);}*Oltre a eliminare il superfluo *overload* di *funza* con la segnatura *funza(int const &)*, in questa versione il valore restituito dalla prima esecuzione di *funza* è stato assegnato, per immediata inizializzazione, a un identificatore *c* esplicitamente dichiarato, nell'ambito di *main*, come riferimento a *rvalue*. Per inciso, una tale dichiarazione esplicita non può prescindere da un'immediata inizializzazione, vale a dire che una dichiarazione pura del tipo *int && c;* è considerata un errore.  
In seguito, come ultima istruzione di *main*, si è aggiunta un'ulteriore invocazione di *funza*, trasferendole proprio *c* come argomento. In *output* appare quanto segue, sempre eseguendo senza alcuna stringa aggiuntiva sulla linea di esecuzione:  
  
*sono funza(int &) 1  
sono funza(int &&) 5  
sono funza(int &&) 8  
sono funza(int &&) 8  
sono funza(int &) 1*vale a dire che *c* è considerato un *lvalue*!  
Su questa regola, ossia sulla proprietà di un riferimento a *rvalue* di poter essere trattato da *lvalue*, si tornerà più approfonditamente in seguito.  
  
Riassumendo le regole di concomitanza per gli *overload* di funzioni, limitandosi, senza alcuna perdita di generalità, a quelle che ricevono un solo argomento, di tipo *int* (per ogni altro tipo e/o altro numero di argomenti valendo identiche considerazioni), si può affermare quanto segue:

1. se si definisce la segnatura *funza(int)* nessun altro *overload* è compatibile (*funza(int \*)* va trattata come ricevente un tipo diverso); alla funzione possono essere trasmessi sia *lvalues* sia *rvalues* che passano nell'ambito della funzione come copie e diventano comunque, localmente, *lvalues*: è il caso, mantenuto, del C ANSI.
2. se si definisce *funza(int &)*, non è più possibile, come si è appena detto, definire *funza(int)*; se non si definisce alcun altro *overload*, alla funzione possono essere trasmessi solo *lvalues*, MAI *rvalues*.
3. se si definisce solo *funza(int const &)* si possono trasmettere indifferentemente *lvalues* e *rvalues*, ma la funzione non si deve permettere di modificarli internamente, nemmeno se sono *lvalues*.
4. se si definiscono sia *funza(int &)* sia *funza(int const &)* (che sono perfettamente compatibili l'uno con l'altro) il primo sarà usato quando si trasmette un *lvalue*, che potrà essere modificato, e il secondo quando si trasmette un *rvalue*, che NON dovrà essere modificato.
5. se si definisce SOLO *funza(int &&)* alla funzione possono essere trasmessi SOLO *rvalues*, NON *lvalues*. Tuttavia l'argomento ricevuto PUÒ ESSERE MODIFICATO nell'ambito della funzione: è questa un'altra conferma dell'annunciata regola secondo cui un riferimento a *rvalue* può essere trattato come un *lvalue*.
6. se si definiscono sia *funza(int &)* sia *funza(int &&)* il primo riceverà solo *lvalues* e il secondo solo *rvalues*, ma tutti e due potranno modificare internamente l'argomento ricevuto, a conferma di quanto già detto.
7. se si definiscono sia *funza(int const &)* sia *funza(int &&)*, ma non *funza(int &)*, il primo *overload* si curerà di ricevere *lvalues*, SENZA MODIFICARLI, e il secondo riceverà *rvalues*, POTENDOLI MODIFICARE; un caso paradossale in cui *lvalues* e *rvalues* si scambiano, apparentemente, una delle loro più significative caratteristiche.
8. se si definiscono tutti e tre gli *overload* si ricade nel caso 6. perché la segnatura *funza(int const &)* viene oscurata da *funza(int &)* per quanto riguarda gli argomenti *lvalue* e da *funza(int &&)* per quelli *rvalue*.

**Gli *overload* forniti gratuitamente**

Benché, come si è appena finito di dire, la segnatura per cui un argomento ricevuto sia un riferimento a *rvalue* si possa applicare a qualsiasi funzione, il protocollo della *move semantic* nasce essenzialmente, come detto all'inizio del terzo paragrafo di questo stesso documento, per modificare il comportamento del costruttore di copie e dell'operatore di assegnamento; al punto che il compilatore stesso genera *sua sponte*, per qualsiasi *class* o *struct* introdotta dal programmatore, un *move constructor* e un *move assignment operator*, come ampiamente discusso nel canto precedente.  
Richiamando alla memoria il punto 7 del paragrafo precedente si vede che i due *overload* impliciti dell'operatore di assegnamento e i due costruttori impliciti NON *default* ricadono in quella casistica.

**La funzione *move* e la *move semantic* forzata**

Nel *namespace* **std** lo *standard* 2011 ha introdotto un numero *N* di nuove funzioni confrontabile col numero di Avogadro. Tra queste vi è la funzione *move* che è una funzione *template* sulla cui segnatura, per il momento, conviene soprassedere. Il suo uso consente di invocare il protocollo della *move semantic* anche quando non sarebbe utilizzato automaticamente; si considerino le seguenti linee di codice, in cui tutti gli identificatori presenti si sottintendono essere oggetti istanziati di una certa classe *Ciccio*:  
  
*a = b;  
c = std :: move(d);*Tutti e quattro gli identificatori *a*, *b*, *c* e *d* sono, per natura, *lvalues*; pertanto nella prima linea di codice si invoca l'operatore di assegnamento, implicito o definito, *copy*. Nella seconda, invece, si invoca l'operatore di assegnamento *move* perché la funzione *std::move* ha appunto l'effetto di restituire un riferimento a *rvalue* relativo al proprio argomento (GRAN LINGUAGGIO). Come ci riesca è evidentemente nascosto nelle pieghe della sua definizione, e si tornerà su questo fra breve; qui è rilevante domandarsi che cosa comporti quella seconda linea di codice, in cui l'oggetto *c* e l'oggetto *d* si scambiano i propri contenuti, diversamente da quanto accade nella linea precedente, in cui è verosimile che vada definitivamente perduto il precedente contenuto di *a*.   
**Quando va perduto il precedente contenuto di *c*?**  
La risposta più ovvia sarebbe: *"quando sarà distrutto l'oggetto d, che lo contiene"*...ma *d* è un *lvalue* tuttora presente nello stesso ambito di visibilità...se andasse incontro a ulteriori *move*, magari transitando in qualche funzione invocata successivamente in quello stesso ambito? Il contenuto originario di *c* (non ci si dimentichi che di QUELLO si tratta) rischia di sfuggire alla gestione del programma, come una piuma portata dal vento: prima o poi finirà sotto il maglio del distruttore di *Ciccio*, ma se ciò accadesse in un punto e in un momento sfavorevoli, perché l'azione del distruttore ha effetti collaterali sul programma che, a quel punto, sono usciti di controllo?  
  
Sono queste le ragioni per le quali, quando si volesse forzare l'uso della *move semantic*, come nel caso presente, ma si avessero effetti collaterali connessi alla distruzione dei propri oggetti, è altissimamente consigliabile, direi quasi irrinunciabile, non affidarsi agli *overload* impliciti, ma assumersene la responsabilità personale scrivendo le proprie funzioni *operator = (Ciccio &&)*, o *Ciccio(Ciccio &&)*, e operando al loro interno in modo adeguato a coprirsi le spalle per qualsiasi evenienza. Dopo tutto il motto del comitato di redazione dello *standard* C++ recita, più o meno, che gli sviluppatori del linguaggio non vi introdurranno mai caratteristiche grammaticali tali da impedire a un programmatore, che lo volesse, di spararsi nel proprio alluce. In altre parole aderiscono, come l'autore di queste note, alla scuola di pensiero draconiana: chi sbaglia, paghi.

**La regola *"se ha un nome, allora è un lvalue"***

Si è più volte accennato, in questo stesso documento, al fatto che un riferimento a *rvalue* possa essere trattato come un *lvalue*. Il titolo del presente paragrafo enuncia appunto la regola fondamentale per potere stabilire quando ciò possa verificarsi. Diamo qui di seguito alcuni esempi che servano a chiarire il significato di questa regola.

* *class Ciccio {/\* omissis \*/};  
  void funza(Ciccio && ciccio)  
  {/\* in funza ciccio è un lvalue; infatti ha un nome, pertanto... \*/  
  Ciccio coccio(ciccio); // invoca il costruttore di copie Ciccio(const Ciccio &)  
  }*
* *class Ciccio {/\* omissis \*/};  
  Ciccio && funza(  )  
  {/\* omissis, convenendo che funza restituisca al chiamante un riferimento a rvalue\*/}  
    
  int main(  )  
  {Ciccio ciccio(funza(  ));  
  /\* si invoca il move constructor Ciccio(Ciccio&&) perché  
  il riferimento a rvalue restituito da funza non ha un nome   
  nell'ambito di main \*/}*

Dovrebbe essere chiaro che questa regola è fatta apposta per minimizzare gli inconvenienti di cui si è parlato nel paragrafo precedente, perché impedisce, a meno che uno non lo faccia proprio apposta, con la funzione *std::move*, che sia usata la *move semantic* a carico di qualcosa che, avendo un nome, resta visibile e accessibile nell'ambito in cui si trova; e si capisce anche che la funzione *std::move* raggiunge il suo scopo di trasformare il suo argomento in un riferimento a *rvalue* semplicemente togliendogli, ovvero nascondendone in qualche modo, il nome. Si tornerà su questo più avanti, quando si specificherà esplicitamente la segnatura di *std::move*. Una conseguenza formidabile di ciò si ha nel caso delle gerarchie ereditarie di classi; se una *class/struct B* è erede di una *class/struct A*, e se si volesse implementare correttamente la *move semantic* per gli oggetti della classe figlia, si dovrebbe procedere come segue:  
  
*class A { // omissis  
public:  
A(  ) = default; // richiesto costruttore default  
A(const A& a) {/\*costruttore di copie\*/}   
A(A&& a) {/\* move constructor \*/}   
};  
  
class B : public A  
{// omissis  
public:  
B(  ) = default; // richiesto costruttore default  
B(const B& b) : A(b)  
{/\*costruttore di copie che invoca il costruttore di copie di A\*/}  
B(B&& b) : A(std::move(b))   
{/\*move constructor che invoca il move constructor di A\*/}  
};*Se il *move constructor* di *B* avesse invocato il costruttore di *A* trasmettendogli il riferimento a *rvalue b* nudo e crudo sarebbe stato eseguito il costruttore di copie di *A* e NON il suo *move constructor*, spezzando quindi il protocollo della *move semantic*; e questo, ancora una volta, perché il riferimento *b* ha un nome nell'ambito del *move constructor* di *B* e quindi è un *lvalue*. Provare per credere: si esegua un programma contenente il codice precedente, magari inserendo degli ordini di scrittura su *output* che facciano comprendere che cosa viene eseguito, e in quale ordine, quando si istanzino in un ipotetico *main* degli oggetti *B*.

**Il problema del *perfect forwarding***

L'esempio che conclude il paragrafo precedente è una delle manifestazioni del cosiddetto problema del *perfect forwarding*, ossia del miglior modo di trasmettere a una funzione un parametro che essa, a sua volta, debba ritrasmettere a un'altra funzione invocata dal proprio interno, così che quest'ultima lo riceva nel modo più appropriato al suo buon funzionamento: è esattamente quanto si auspicava che accadesse rispetto all'invocazione del costruttore della classe antenata.  
Nella sua forma più generale il problema si pone nei seguenti termini: si supponga di scrivere una funzione che debba fungere da involucro per un'altra; la necessità, o l'opportunità, di introdurre un simile involucro può dipendere da numerosi fattori, uno dei più frequenti essendo dato dal fatto che le funzioni coinvolte siano delle funzioni *template*. Sia dunque data una funzione *template* così fatta:  
  
*template <typename T, typename A> T \* allocca(A a)  
{return new T(a);}*   
  
che ha lo scopo evidente e lodevole di inizializzare un puntatore a un tipo generico *T* che abbia un costruttore parametrico dipendente da un parametro, altrettanto generico, appartenente a un tipo *A* (GRAAAAN LINGUAGGIO...)  
Purtroppo gli eccessivi entusiasmi per avere escogitato questa brillante soluzione sono frustrati dal fatto che la soluzione non è affatto brillante dato che il *forwarding* dell'argomento *a* al costruttore *T(a)* è ben lungi dall'essere *perfect* dato che *a* è ricevuto per copia da *allocca* e il costruttore potrebbe anche voler ricevere per indirizzo il proprio argomento.  
Né vale far ricevere *a* da *allocca* per indirizzo, o come *A const &*, o come *A &&*, visto tutto quello che si è detto nei paragrafi precedenti: in ogni caso ci può essere una controindicazione, da parte del costruttore *T*, per rendere imperfetto il *forwarding* di *a*. E allora? Bisogna quindi rinunciare alla brillante idea di realizzare una sorta di allocatore universale? La risposta è, ovviamente, NO, perché il linguaggio è davvero un gran linguaggio e, per questa bisogna, ha introdotto nello *standard* 2011 due ulteriori nuove regole, impensabili prima, che sono inserite in questo stesso documento, piuttosto che in uno proprio, data la loro stretta attinenza con la *move semantic*.

**La regola del collasso degli &**

Si riassume in poche righe; per qualsiasi tipo *T*:  
*T&    &* si interpreta come *T &  
T&    &&* si interpreta come *T &  
T&&    &* si interpreta come *T &  
T&&    &&* si interpreta come *T &&*e ricorsivamente.

**La regola di deduzione del riferimento *template* a *rvalue***

Se una funzione *template* ha un argomento che ha come tipo un riferimento a *rvalue* del tipo templatizzato, ossia, per intendersi, se una funzione *template* è così dichiarata (che sia usato il tipo restituito *void* è irrilevante):  
 *template <typename T> void funza(T&&);*   
  
allora, ogni volta che si invochi *funza* trasferendole un *lvalue* di tipo *A* il tipo *T* si istanzia in *A&* e quindi, in base alla regola precedente, l'argomento viene ricevuto come *A&*.  
Se invece a *funza* si trasferisce un *rvalue* di tipo *A*, allora *T* si istanzia semplicemente in *A* e quindi l'argomento viene ricevuto come un riferimento a *rvalue*.

**Il *perfect forwarding* realizzato!**

Ecco come va risolto il problema del *perfect forwarding*, riferendosi allo stesso caso del precedente tentativo e tenendo conto di quanto detto nel frattempo:  
  
*template <typename T, typename A> T \* allocca(A&& a)  
{return new T(std::forward<A>(a));}*   
  
ove si è fatto uso della funzione *template* chiamata *forward*, appositamente definita per questa bisogna, nel *namespace* **std**, come segue:  
  
*template <class S> S&&  
forward(typename remove\_reference<S>::type& a) noexcept  
{return static\_cast<S&&>(a);}*e la *struct* *template* chiamata *remove\_reference* è stata presentata e discussa nel documento su *typedef*.  
Per convincersi che questo *ambaradan* funziona in ogni caso, si allega il seguente codice esemplificativo:  
  
*# include <iostream>  
using namespace std;  
  
struct Pappo  
{Pappo(  ) = default;  
Pappo(int i) {cout << "sono Pappo(int " << i << ")\n";}  
Pappo(double i) {cout << "sono Pappo(double " << i << ")\n";}};  
  
struct Peppo  
{Peppo(  ) = default;  
Peppo(int i) {cout << "sono Peppo(int " << i << ")\n";}  
Peppo(double i) {cout << "sono Peppo(double " << i << ")\n";}};  
  
class Ciccio  
{public:  
Ciccio(int i) {cout << "sono Ciccio(int)\n";}  
Ciccio(char \*i) {cout << "sono Ciccio(char \*)\n";}  
Ciccio(Pappo &p) {cout << "sono Ciccio(Pappo&)\n";}  
Ciccio(Peppo &&p) {cout << "sono Ciccio(Peppo&&)\n";}  
Ciccio(Pappo &&p) {cout << "sono Ciccio(Pappo&&)\n";}  
Ciccio(Peppo &p) {cout << "sono Ciccio(Peppo&)\n";}};  
  
template <typename T, typename A> T \* allocca(A&& a)  
{return new T(forward<A>(a));}  
  
int main(  )  
{auto \* pappo\_int = allocca<Pappo>(1), \* pappo\_double = allocca<Pappo>(1.5);  
auto \* peppo\_int = allocca<Peppo>(2), \* peppo\_double = allocca<Peppo>(2.5);  
auto  
\* ciccio\_int = allocca<Ciccio>(1),  
\* ciccio\_pappo\_lvalue = allocca<Ciccio>(\* pappo\_int),  
\* ciccio\_peppo\_rvalue = allocca<Ciccio>(move(\* peppo\_double));  
// e così via  
}*A parte lo scontato riconoscimento automatico delle istanze, proprio di ogni funzione *template*, la parte davvero interessante del precedente codice sta nelle due ultime inizializzazioni dei puntatori *ciccio\_pappo\_lvalue* e *ciccio\_peppo\_rvalue*, che avvengono attraverso gli appropriati costruttori, nonostante l'intermediazione di *allocca*, in virtù, per l'appunto, del conseguito *perfect forwarding*. In effetti basta seguire, alla luce delle regole citate, il destino dell'argomento trasferito ad *allocca* per capacitarsi di come il programma si comporti.

**come è fatta la funzione *move***

Ecco qui la definizione della funzione *move* così come si trova nel *namespace* **std**:  
  
*template <class X>  
typename remove\_reference<X>::type&&  
std::move(X && a) noexcept  
{ typedef typename remove\_reference<X>::type&& RR;  
return static\_cast<RR>(a);}*Qui *remove\_reference* è la classe *template* descritta alla fine del documento su *typedef* e, se si tiene conto di tutto quanto detto fin qui, in questo stesso documento, si dovrebbe capire come il tutto funziona.  
In effetti, se si invoca *move* trasferendole un *lvalue* di tipo *A*, come nella sequenza  
  
*A a;  
std::move(a);*allora, in base alla regola di deduzione del tipo templatizzato, *move* risulta istanziata nella seguente forma  
  
*typename remove\_reference<A&>::type&&  
std::move(A& && a) noexcept  
{ typedef typename remove\_reference<A&>::type&& RR;  
return static\_cast<RR>(a);}*e, dopo aver valutato l'effetto di *remove\_reference* e aver applicato la regola per il collasso degli &, ciò che il compilatore, *de facto*, si trova a dover compilare è la seguente versione finale di *move*:   
  
*A&& std::move(A& a) noexcept  
{return static\_cast<A&&>(a);}*che è la cosa giusta che deve accadere. Altrettanto giusto sarebbe ciò che accadrebbe, in base alle stesse regole, se a *move* si trasferisse un *rvalue* invece che un *lvalue*, quantunque verrebbe spontaneo domandarsi per quale diamine di motivo uno dovrebbe mandare un *rvalue* a *move*, mettendosi da solo nei guai per quanto attiene la distruzione dell'oggetto trasferito.  
Dalla forma di *move* si possono trarre un paio di deduzioni:

1. al suo posto sarebbe altrettanto valido l'uso di *static\_cast*
2. quando si scrivono funzioni che hanno argomenti di tipo *riferimento a rvalue* lo *standard* raccomanda caldissimamente l'uso della parola di vocabolario *noexcept*, e questo per evitare conflitti tra la *move semantic* e certe funzioni della *standard template library*, come ad esempio il ridimensionamento di un oggetto *vector*, per le quali la *move semantic* non sarebbe applicata, neppure se implementata, in assenza di *noexcept*.

**Canto trentasettesimo: le espressioni *lambda***

Con questo termine si indicano essenzialmente delle funzioni anonime che:

* possono essere definite anche all'interno dell'ambito di un'altra funzione, oltre che nell'ambito globale;
* possono essere assegnate a un oggetto di tipo *std::function* (una classe *template* definita nel *namespace* **std**), acquisendo in tal modo un nome;
* possono essere trasmesse come argomento ad altre funzioni, pur rimanendo, in tale contesto, senza nome;
* possono gestire variabili dichiarate nell'ambito in cui esse stesse sono definite.

La sintassi minimale di un'espressione *lambda* inutile, perché nullafacente, è la seguente:  
  
*[  ]{  }*  
  
La precedente sequenza di caratteri, ovviamente terminata col segno di punto e virgola, o con l'operatore virgola, può apparire in qualsiasi ambito di funzione, NON in un ambito di classe o in un *namespace* o nell'ambito globale; in questi ambiti può apparire solo se assegnata a un identificatore dichiarato, come ad esempio in  
  
*auto u = [  ]{  };*   
  
La coppia di parentesi quadre, fin qui vuota, è chiamata specificatore di cattura e deve sempre essere presente, anche se vuota; la coppia di parentesi graffe costituisce, come d'abitudine, il corpo, ossia l'ambito, dell'espressione e appare del tutto ovvio che debba esserci.  
  
Tra la coppia di quadre e la coppia di graffe vi può essere una coppia di parentesi tonde destinata a contenere la lista degli argomenti, come accade per ogni buona funzione; tuttavia la sua presenza non è obbligatoria quando tale lista dovesse essere vuota. Qui di seguito si dà un esempio di due dichiarazioni completamente equivalenti di due espressioni *lambda* all'interno dell'ambito di *main*:   
  
*int main(  )  
{auto u = [  ] {std :: cout << "io sono un'espressione lambda\n";};  
auto v = [  ] (  ) {std :: cout << "io sono un'espressione lambda\n";};}*   
  
Nel precedente esempio ai due oggetti *u* e *v* è assegnata, per immediata inizializzazione, la stessa espressione, quantunque scritta in due modi diversi. Il tipo comune dei due oggetti è arguito dal compilatore stesso, grazie alla parola di vocabolario *auto*. Se si compila ed esegue il precedente programma, tuttavia, non si vedrà accadere un bel nulla: gli oggetti dichiarati e inizializzati non sono mai stati eseguiti. Affinché accada qualcosa di concreto occorre modificare il codice, ad esempio così:  
  
*int main(  )  
{auto u = [  ] {std :: cout << "io sono un'espressione lambda\n";};  
auto v = [  ] (  ) {std :: cout << "io sono un'espressione lambda\n";};  
u(  ), v(  );}*   
  
In questo caso si vedrà apparire due volte, su *output*, la frase  
*io sono un'espressione lambda*  
una dovuta all'esecuzione dell'oggetto *u*, l'altra a quella dell'oggetto *v*. Appare evidente che i due oggetti agiscono, a tutti gli effetti, come delle funzioni: se ne invoca l'esecuzione con la sintassi tipica dell'invocazione di una funzione e l'ambito della funzione è stato inserito direttamente entro *main* e non all'esterno: non esistono due funzioni con tali nomi dichiarate nell'ambito globale.  
  
Si è già detto che entro la coppia di parentesi tonde si possono inserire argomenti per l'espressione *lambda* i quali, come accade per tutte le funzioni normali, transitano all'interno dell'ambito dell'espressione. Ecco una variante del codice precedente, in cui si fa uso di un argomento, e che produce lo stesso identico risultato  
  
*int main(  )  
{auto u = [  ] {std :: cout << "io sono un'espressione lambda\n";};  
auto v = [  ] (const char \* s = "io sono un'espressione lambda\n") {std :: cout << s;};  
u(  ), v(  );}*   
  
Si osservi che, rispetto all'invocazione di una funzione ordinaria, l'argomento fornito può essere dichiarato e inizializzato direttamente nella lista, e che l'invocazione dell'oggetto non necessita del reinserimento dell'argomento. Per rendere più chiara quest'ultima affermazione si consideri quest'ulteriore variante:  
  
*int main(  )  
{const char \* t = "tanto va la gatta al lardo";  
auto u = [  ] {std :: cout << "io sono un'espressione lambda\n";};  
auto v = [  ] (const char \* s = "io sono un'espressione lambda\n") {std :: cout << s;};  
u(  ), v(  ), v(t);}*   
  
in cui l'oggetto *v* è eseguito due volte, una volta senza inserire alcun argomento nell'invocazione e un'altra inserendo un argomento, ovviamente di tipo compatibile; come ormai ci si dovrebbe attendere l'*output* sarà  
  
*io sono un'espressione lambda  
io sono un'espressione lambda  
tanto va la gatta al lardo*vale a dire che se l'argomento è inizializzato direttamente nella dichiarazione funge, a TUTTI gli effetti, da argomento *standard*, come accade per qualsiasi funzione.  
  
Si è detto, all'inizio di questo documento, che le espressioni *lambda* possono essere trasferite, così come sono state fin qui scritte, ad altre funzioni, come argomenti, beninteso purché le funzioni destinatarie siano predisposte a riceverle: si tratta quindi di un meccanismo alternativo a quello del trasferimento di puntatori a funzione, per consentire alla funzione destinataria di poter ricevere non solo dati, ma anche istruzioni da eseguire.  
Ecco un esempio minimale in cui una funzione ordinaria, la famigerata *funza*, riceve come argomento un'espressione *lambda*:  
  
*# include <iostream>  
# include <functional>  
using namespace std;  
  
void funza(function<void(  )> f)  
{f(  );}  
  
int main(  )  
{funza([  ]{cout << "la gloria di Colui che tutto move\n";});}*   
  
Nel precedente programma sono da notare:

* l'invocazione di *funza* da parte di *main*, trasferendole come argomento direttamente un'espressione *lambda* non assegnata ad alcunché e quindi completamente anonima (cfr. anche il canto precedente);
* la dichiarazione di *funza*, come funzione atta a ricevere un unico argomento che sia un'istanza della *template* class *function<void(  )>*: questo è, in definitiva, il tipo attribuito all'espressione *lambda* trasferita a *funza*;
* l'invocazione, da parte di *funza*, dell'esecuzione dell'argomento ricevuto; questo produce l'esecuzione della nostra espressione *lambda* e la comparsa su *output* dell'immortale primo verso della terza cantica del Divino Poema;
* l'inclusione del file *functional*, che consente l'uso della *template* class *function*, ivi dichiarata nell'ambito del *namespace* **std**.

Quanto detto nel secondo punto del precedente elenco comporta che agli oggetti *u* e *v* dichiarati nei primi esempi con l'ausilio della parola di vocabolario *auto* venissero attribuiti tacitamente, da parte del compilatore, i tipi *function<void(  )>* fino al momento del trasferimento di un argomento all'oggetto *v* cui veniva, per questa ragione, attribuito il tipo *function<void(const char \*)>*.  
  
**Osservazione**  
  
Trasferire a una funzione, come argomento, un'espressione *lambda*, come si è appena visto, È TOTALMENTE DIVERSO dal trasferirle un puntatore a funzione: per persuadersene basta leggere la dichiarazione di *funza* nell'esempio. Tuttavia, trasformando *funza* in una funzione *template*, si ottiene il non trascurabile vantaggio di poterle trasferire indifferentemente o l'una o l'altro. La seguente variante del codice precedente illustra l'asserto (GRANDISSIMO LINGUAGGIO)  
  
*# include <iostream>  
# include <functional>  
using namespace std;  
  
/\*1\*/typedef void (\*f)(  );  
  
f     punt;  
  
void funzb(  )  
{cout << "per l'universo penetra, e risplende\nin una parte più e meno altrove\n";}  
  
/\*2\*/template <typename tipo> void funza(tipo f)  
{f(  );}  
  
int main(  )  
{funza([  ]{cout << "la gloria di Colui che tutto move\n";});  
funza(punt = funzb);}*  
  
Nella riga di codice */\*1\*/* si usa *typedef* per attribuire a *f* il significato di tipo per un puntatore a funzione capace di indirizzare una funzione *void* senza argomenti, attribuzione che viene immediatamente sfruttata, nella riga successiva, per dichiarare un tale puntatore, chiamato *punt*.  
Nella riga di codice */\*2\*/* si trasforma *funza*, come anticipato, in una funzione *template*; a quel punto *main* la istanzia/invoca due volte trasferendole la prima volta un'espressione *lambda* e la seconda volta il puntatore *punt*, inizializzato a volo nella lista stessa di invocazione di *funza* (GRAN LINGUAGGIO). Tutto avviene in modo silente e trasparente e il risultato netto è il completamento, su *output*, della prima terzina dell'ultima cantica del *"Poema Sacro, al quale ha posto mano e Cielo e Terra"* (cit. dal medesimo).   
  
Fin qui si è parlato di espressioni *lambda* equivalenti a funzioni che non restituiscono nulla al chiamante, vale a dire, per intendersi, di tipo *void*; questo è ciò che il compilatore assume, senza ulteriori perdite di tempo, quando, come è sempre accaduto fin qui, nell'ambito dell'espressione non appare mai l'istruzione *return*. Se questa dovesse apparire, ma non più di una sola volta, il compilatore continuerebbe a poter arguire il tipo restituito semplicemente esaminando quello dell'oggetto coinvolto nella singola istruzione *return* e tutto proseguirebbe come finora: un eventuale oggetto cui venisse assegnata una tale espressione *lambda* potrebbe ancora essere dichiarato come di tipo *auto* e diverrebbe automaticamente un'istanza di *function<T(/\*omissis\*/)>* ove *T* è il tipo riscontrato dell'oggetto restituito e al posto di */\*omissis\*/* va l'intera, eventuale, lista dei tipi degli argomenti trasmessi all'espressione.  
  
In casi più complessi, in cui ci sia più di un solo *return* o comunque se il programmatore intende mettere in evidenza il tipo restituito dall'espressione in modo coerente, beninteso, con quanto poi viene effettivamente inserito a destra di *return*, desso tipo restituito può essere inserito *prima dell'ambito dell'espressione* utilizzando la sintassi introdotta con lo *standard* 2011, come nelle due seguenti dichiarazioni:  
  
*auto u = [  ] (/\*omissis\*/) -> int {/\*omissis\*/ return 1;}  
auto v = [  ] (int k, double l) -> decltype(l) {/\*omissis\*/ return 1.5;}*   
  
In questi due esempi *u* diventa un'istanza di *function<int(/\*omissis\*/)>* e *v* un'istanza di *function<double(int,double)>*. Non dovrebbe essere necessario sottolineare che *function* è una classe *template* con numero variabile di tipi templatizzati.   
  
Va sottolineato che un'espressione *lambda* può anche essere eseguita al volo, ossia all'atto stesso della sua definizione. In altri termini la seguente variante dell'ultimo programma qui presentato  
  
*# include <iostream>  
using namespace std;  
  
int main(  )  
{[  ]{cout << "la gloria di Colui che tutto move\n";}(  );}*   
  
in cui va apprezzata l'aggiunta delle due parentesi tonde a destra, che implicano l'invocazione dell'esecuzione, benché sia del tutto oziosa, produce lo stesso risultato su *output*. Naturalmente, però, un'espressione *lambda* definita ed eseguita simultaneamente, senza essere stata assegnata ad alcun oggetto e rimasta quindi anonima, non può essere rieseguita una seconda volta, se non a prezzo di riscriverla.   
  
Un'espressione *lambda* può comparire anche all'interno di una *class/struct* ed essere assegnata a un suo membro, opportunamente dichiarato. In tal caso, però, non può accedere ad altri membri dell'oggetto cui appartiene che NON siano stati qualificati con la parola di vocabolario *static*, né può essere a sua volta qualificata come *const* o *volatile*, diversamente dai metodi membri ordinari. Ecco un esempio, in cui è sottoposto a commento ciò che è vietato:  
  
*# include <iostream>  
# include <functional>  
using namespace std;  
  
struct Ciccio  
{constexpr double const static c = 2.12;  
int d = 5;  
function<void(  )> u = [  ] /\*const\*/ {cout << c << endl; /\*cout << d << endl;\*/};};   
  
int main(  )  
{Ciccio c;  
c . u(  );}*   
  
Peraltro è sempre consentito a un'espressione *lambda*, sia membro di classi o no, specificare le eccezioni che le è permesso lanciare; nella seguente variante del programma precedente si specifica che l'espressione *lambda* assegnata all'oggetto membro *u* non può lanciare alcuna eccezione, cosa che, del resto, non fa.  
  
*# include <iostream>  
# include <functional>  
using namespace std;  
  
struct Ciccio  
{constexpr double const static c = 2.12;  
function<void(  )> u = [  ] (  ) throw(  ) {cout << c << endl;};};   
  
int main(  )  
{Ciccio c;  
c . u(  );}*   
  
Si osservi il seguente particolare: quando si inserisce la parola *throw*, la lista degli argomenti vuota passa dall'essere opzionale al diventare obbligatoria; quindi, visto che costa così poco metterla, perché non metterla sempre?  
  
Se un/a lettore/trice di queste note non si è ancora addormentato/a, si dovrà pur domandare, prima o poi, a che diamine serve quello che è stato chiamato fin dall'inizio specificatore di cattura, ossia la coppia di parentesi quadre, e che finora è sempre stato lasciato vuoto, pur dovendo obbligatoriamente essere SEMPRE presente.  
La cattura di cui si parla riguarda le variabili dichiarate nello stesso ambito in cui è dichiarata l'espressione *lambda* stessa (vedere l'ultimo punto dell'elenco con cui questo documento esordisce).  
Quando la coppia di quadre è vuota, NESSUNA di tali variabili è accessibile all'espressione *lambda*; nel suo ambito entrano SOLO gli eventuali argomenti trasmessi entro le parentesi tonde. Questo, a ben pensarci, è il comportamento abituale di QUALSIASI funzione.  
Ma allorché entro le parentesi quadre si mette qualcosa, questo genera una via alternativa e complementare al trasferimento di argomenti per consentire alle espressioni *lambda* qualche forma di accesso alle variabili dichiarate nel loro stesso ambito. Il seguente programma illustra probabilmente TUTTI i casi possibili:  
  
*# include <iostream>  
# include <functional>  
using namespace std;  
  
class Ciccio  
{int i = 1;  
double d = 2.4;  
public:  
void funza(  )  
{auto u = [this] {cout << i << ' ' << d << endl;};  
u(  );  
[this] {cout << i << ' ' << d << endl;}(  );}} c;  
  
struct Coccio  
{static const int i = 111;  
constexpr static double d = 222.4;  
function<void(  )> u = [this] {cout << i << ' ' << d << endl;};} d;  
  
int main(  )  
{int c = 90;  
double r = 7.7;  
/\*1\*/[c] (double d) {cout << c << ' ' << d << endl;}(6.4);  
auto u = [c] (double d) {cout << c << ' ' << d << endl;};  
/\*2\*/u(r);  
auto v = [c] (double c) {cout << c << endl;};  
/\*3\*/v(r);  
auto w = [c,r] {cout << c << ' ' << r << endl;};  
/\*4\*/w(  );  
auto x = [&c,r] {cout << ++c << ' ' << r << endl;};  
/\*5\*/x(  );  
cout << "in main c = " << c << endl;  
auto y = [&,r] {cout << ++c << ' ' << r << endl;};  
/\*6\*/y(  );  
cout << "in main c = " << c << endl;  
auto z = [=] (double s, int u) {cout << s << ' ' << u << ' ' << r << endl;};  
/\*7\*/z(1.2, 12);  
/\*8\*/[&] {cout << ++c << ' ' << (r += 1.0) << endl;}(  );  
cout << "in main c = " << c << "; r = " << r << endl;  
/\*9\*/[=,&r] (const char \* w)  
{cout << c << ' ' << (r \*= 1.4) << ' ' << w << endl;}  
("Per correr miglior acque alza le vele");  
cout << "in main r = " << r << endl;  
/\*10\*/:: c . funza(  );  
/\*11\*/d . u(  );  
/\*12\*/[=](  ) mutable {cout << "modifico c " << ++c << endl;} (  );}*   
  
Nel codice appena presentato si trova un gran numero di espressioni *lambda*, NESSUNA delle quali ha lo specificatore di cattura vuoto. Traendo giovamento dai numeri inseriti come commenti si darà, qui appresso, una spiegazione del tutto.

* in */\*1\*/* viene definita ed eseguita a volo un'espressione *lambda* che riceve un argomento *double* e alla quale si consente l'accesso alla variabile intera *c* dichiarata in *main*, avendone posto il nome entro le parentesi quadre. All'atto dell'esecuzione viene assegnato all'argomento il valore 6.4; il risultato netto è che in *output* appare la linea  
  90  6.4
* in */\*2\*/* viene eseguita, attraverso l'oggetto *u* cui era stata assegnata nella riga precedente, la stessa espressione eseguita in */\*1\*/*, ma stavolta trasmettendole come argomento la variabile *r* di *main*: il risultato in *output* è la linea  
  90  7.7
* in */\*3\*/* viene eseguita un'espressione in cui si vede che cosa succede in caso di omonimia, che il compilatore consente, tra un argomento e una variabile che appare nello specificatore di cattura; secondo il compilatore GNU-gcc è quest'ultima a prevalere e l'argomento di tipo compatibile che deve comunque essere trasmesso, per consentire l'esecuzione dell'espressione, non potrà MAI essere utilizzato, perché coperto dalla variabile esterna. In *output* appare dunque la linea  
    
  90  
    
  Ciò sarebbe CONTRADDITTORIO rispetto al comportamento delle funzioni normali, per le quali è sempre l'argomento ricevuto a prevalere su un'eventuale variabile omonima dichiarata in un ambito esterno (quello globale, per le funzioni ordinarie). Pertanto questa caratteristica del compilatore GNU-gcc, secondo lo stesso parere degli esperti dell'*help desk* su internet, cui la questione è stata sottoposta dall'autore di questi canti nel luglio 2014, va considerata erronea e sarà verosimilmente emendata in una ventura versione del compilatore. Peraltro, nel frattempo, uscirà anche lo standard 2014 del linguaggio che, auspicabilmente, farà maggiore chiarezza su questo punto.   
  La morale di tutto questo sproloquio è che, siccome il numero di nomi diversi che sono disponibili ha un numero di cifre decimali dell'ordine di dieci alla cinque (ALMENO), non si vede per quale motivo ci si dovrebbe ridurre a cadere in simili situazioni di omonimia.
* in */\*4\*/* si ottiene lo stesso risultato di */\*2\*/*, con un'espressione SENZA argomenti; ciò è possibile perché all'oggetto *w* è stato consentito di accedere a entrambe le variabili *c* e *r* dichiarate in *main*; e come si accede a DUE variabili si accederà a un numero arbitrario...
* in */\*5\*/* si ha la stessa situazione di */\*4\*/*, ma stavolta all'oggetto *x* è stato concesso di accedere alla variabile *c* col DIRITTO DI MODIFICARLA, diritto che non è ammesso a proposito di *r*. Ecco perché l'espressione può applicare impunemente l'operatore *++* a *c*. In TUTTI i casi precedenti, e anche in QUESTO caso, relativamente a *r*, ogni pretesa di modifica sarebbe stata segnalata come errore da parte del compilatore. Ciò è profondamente DIVERSO da quanto accade in una funzione ordinaria a proposito degli argomenti ricevuti per valore, che possono essere tranquillamente modificati nell'ambito della funzione; qui, NON trattandosi di argomenti trasferiti, NON VIENE FATTA NESSUNA COPIA, ma semplicemente si concedono diritti a carico di una variabile che è comunque trattata sempre in originale. In *output* si troverà la linea  
  91  7.7  
  seguita da un'altra in cui *main* conferma di essersi ritrovata incrementata la sua variabile *c*.
* Per il caso presente */\*6\*/* è identico a */\*5\*/*; ma inserire nello specificatore di cattura l'operatore & seguito da NULLA significa che si concede diritto di accesso e modifica su TUTTE le variabili esterne, ECCETTO quelle eventualmente nominate esplicitamente di seguito, per le quali pure si concede accesso, ma SENZA diritto di modifica. In *output* apparirà quindi la linea  
  92  7.7  
  ancora una volta seguita da una presa d'atto da parte di *main*.
* In */\*7\*/* si esegue un'espressione che richiede due argomenti in ingresso, uno *double* e uno *int*, cui sono attribuiti rispettivamente i valori 1.2 e 12, e che ha diritto di accesso, SENZA DIRITTO DI MODIFICA, a TUTTE le variabili dichiarate in *main*. Non ci sarà, si spera, chi non si accorga che *[=]* è una provvida scorciatoia che sostituisce l'elenco di tutti i nomi delle variabili, separati da virgole, di cui si era già dato un campione nel caso */\*4\*/*. In *output* si otterrà la linea  
  1.2  12  7.7
* In */\*8\*/* si esegue a volo, in virtù della coppia di parentesi tonde conclusiva, un'espressione che, dal punto di vista dell'accesso alle variabili di *main*, si trova agli antipodi di */\*7\*/* come, a ben guardare, era stato già anticipato in */\*6\*/*: stavolta si ha diritto di ACCEDERE E MODIFICARE sia i cani sia i porci, e in effetti l'espressione non si astiene da nulla, brutalizzando sia *c* sia *r* senza che il compilatore si opponga in alcun modo. In *output* si troverà la linea  
  93  8.7  
  e il *main* ne prende dolorosamente atto nella riga successiva.
* Il commento sugli effetti dell'espressione eseguita a volo in */\*9\*/* è lasciato, come esercizio, a chi legge.
* In */\*10\*/* si richiede l'esecuzione del metodo pubblico *funza(  )* appartenente all'oggetto *:: c* istanziato nell'ambito globale al termine della definizione della *class Ciccio*: ed ecco il motivo della necessità di usare l'operatore ::. Nulla di più normale e abituale, per un programmatore appena più che neofita.  
  Andando però a leggere quello che accade in *funza(  )* si fanno delle scoperte interessanti; un'espressione *lambda*, esattamente la stessa, vi viene assegnata a un oggetto *u*, LOCALE ALLA FUNZIONE, che è poi eseguito, e poi l'espressione viene rieseguita a volo. Che le due esecuzioni producano lo stesso effetto non dovrebbe più essere sorprendente. La cosa rilevante è la presenza di *this* entro lo specificatore di cattura dell'espressione: ciò le consente di citare direttamente le variabili membro private della classe né più né meno come se fosse essa stessa un membro.
* In */\*11\*/* accade una cosa diversa: qui si pretende di eseguire un'espressione *lambda* che sia LEI un membro dell'oggetto *d*. Come si è già visto in questo stesso documento, in un caso simile si deve sottostare ad alcune restrizioni: le variabili membro cui l'espressione pretende di accedere non possono essere qualsiasi, come in */\*10\*/*, ma solo qualificate *static*, a meno che non si provveda alla cattura di *this*, che consente di accedere a TUTTI gli altri membri. Infatti il programma si compila ugualmente ANCHE rimovendo le qualifiche *static* dai membri della classe cui l'espressione *lambda* accede (provare per credere).
* In */\*12\*/*, infine, si vede che l'uso della parola di vocabolario *mutable* consente di neutralizzare l'inserimento di = nello specificatore di cattura, rendendolo equivalente a &.

**Canto trentottesimo: la *standard template library* e gli iteratori**

Si tratta di un'amplissima collezione di farfalle...*pardon*..., di classi e funzioni templatizzate, che, al vostro stato attuale di conoscenza, dovreste sapere scrivere anche voi, ma che, come spesso accade nella vita, qualcun altro ha già scritto e reso disponibili all'*orbe terraqueo*, la qual cosa, se da un lato urta l'autostima del buon programmatore, dall'altro gli risparmia un sacco di fatica e gli garantisce un comportamento, appunto, *standardizzato* delle suddette classi.  
  
Vi si trova di tutto: classi che implementano algoritmi di riordinamento o algoritmi di archiviazione e di ricupero di moli immense di dati, classi per l'internazionalizzazione della tabella dei caratteri, funzioni per l'esecuzione di calcoli numerici e tutto, o quasi, quello che la fantasia del buon programmatore riesce a concepire.   
  
Trattandosi di un numero molto alto, ancorché finito, di classi e funzioni templatizzate, parecchie delle quali variadiche, lascio immaginare a voi, che ormai non potete più essere considerati dei pivelli, QUANTA ROBA CI SAREBBE DA RACCONTARE: così tanta da esaurire il tempo, forse, di un intero corso di Laurea, piuttosto che quello di un insegnamento semiannuale da cinque crediti.  
  
Non potendo, con ogni evidenza, intraprendere una simile impresa, al confronto della quale quella di Sisifo potrebbe essere paragonata al taglio dell'unghia del proprio dito mignolo, ci si limiterà a dire poche cose ed essenzialmente quelle che sono comuni a (quasi) tutte le classi.  
  
Tanto per cominciare diamo alcuni brevi elenchi alfabetici dei loro nomi, NON ESAUSTIVI ma limitati a quelle di uso (forse) più comune; non verrà data neppure la dichiarazione esatta (se vi serve, domandatela a voce) ma solo quale *header* va incluso per poterne fruire assieme a una telegrafica enunciazione dello scopo per cui esistono, così che, quando vi trovaste nella necessità, sappiate che cosa andare a chiedere a qualcuno che già lo sappia:

1. classi template da usare come contenitori di dati; per ciascuna va SEMPRE incluso, salvo avviso contrario, un *header* omonimo:
   * *array* // contenitore di dati di grandezza fissa:
   * *deque* // contenitore di dati dinamico:
   * *forward\_list* // contenitore di dati dinamico:
   * *list* // contenitore di dati dinamico:
   * *map* // contenitore di dati dinamico ordinato associativo:
   * *multimap* // contenitore di dati dinamico ordinato associativo:#include <map>
   * *multiset* // contenitore di dati dinamico ordinato: #include <set>
   * *priority\_queue* // lo dice la parola (cfr. *queue*): #include <queue>
   * *queue* // contenitore di dati che funziona come una coda:
   * *set* // contenitore di dati dinamico ordinato:
   * *stack* // contenitore di dati che funziona come una pila di libri:
   * *string* // contenitore di caratteri:
   * *unordered\_map* // lo dice la parola (cfr. *map*):
   * *unordered\_multimap* // lo dice la parola (cfr. *map*): #include <unordered\_map>
   * *unordered\_multiset* // lo dice la parola (cfr. *set*):#include <unordered\_set>
   * *unordered\_set* // lo dice la parola (cfr. *set*):
   * *vector* // contenitore di dati dinamico:
2. funzioni templatizzate per calcoli numerici: di solito il nome stesso ne spiega il significato; per fruire di queste funzioni occorre includere l'*header* *algorithm* (sovente incluso da *iostream* stesso) e sono TUTTE dichiarate nel *namespace* **std**; quando qui si parla di insieme, generalmente s'intende un'istanza di uno dei contenitori di dati del precedente elenco:
   * *count, count\_if* // restituiscono il numero di elementi di un dato insieme che soddisfanno un assegnato criterio
   * *fill* // assegna un dato valore a TUTTI gli elementi di un dato insieme
   * *find, find\_if, find\_if\_not* // trovano il primo elemento di un insieme che risponda a un dato criterio di ricerca
   * *for\_each* // applica una data funzione a un insieme di elementi
   * *generate* // usa un'assegnata funzione per produrre dati con cui riempire un dato insieme
   * *merge* // produce un unico insieme ordinato a partire da due insiemi ordinati dati
   * *min, max* // restituiscono rispettivamente minimo e massimo di DUE argomenti
   * *partition* // ordina gli elementi di un dato insieme in modo che TUTTI gli elementi che soddisfanno un assegnato criterio precedano TUTTI quelli che non lo soddisfanno.
   * *remove, remove\_if* // elimina da un dato insieme elementi che rispondono a un assegnato criterio, e restituisce un oggetto idoneo a capire quanti sono gli elementi superstiti (che cosa sia un tale oggetto sarà spiegato appresso)
   * *reverse* // inverte l'ordine di un dato insieme
   * *search* // cerca (e magari trova) la prima occorrenza di un dato sottoinsieme in un dato insieme
   * *sort* // ordina in verso ascendente un dato insieme
   * *swap* // scambia i valori dei DUE argomenti che riceve
   * *zzzz\_etcetera* // questa funzione NON C'È: serve solo a terminare l'elenco, ricordando a tutti che ne mancano un sacco e una sporta.
3. funzioni templatizzate per internazionalizzazione della tabella dei caratteri; se ne elencano pochissime dal nome autoesplicativo: per poterne fruire va incluso l'*header* *locale*, e anche queste sono definite in **std**.
   * *isalpha*
   * *isblank*
   * *islower*
   * *isdigit*
   * *ispunct*
   * *isspace*
   * *isupper*
   * *tolower*
   * *toupper*
   * *zzzz\_etcetera*
4. classi e funzioni templatizzate di natura squisitamente matematica, sempre definite in **std**:
   * *complex* // classe templatizzata per i numeri complessi: # include <complex>
   * *valarray* // classe templatizzata contenitore per dati numerici sui cui elementi sono compiute le stesse operazioni: # include <valarray>
   * *ratio* // classe templatizzata per numeri razionali: # include <ratio>
   * *inner\_product* // funzione templatizzata che restituisce il prodotto scalare dei suoi argomenti (tipicamente due istanze di *valarray*, ma non necessariamente [È templatizzata, no?]): # include <numeric>
   * *zzzzz\_etcetera*

Come ben si vede, e come ancor meglio si intuisce, si tratta davvero di qualcosa di confrontabile col Gruppo Locale cui appartiene anche la Via Lattea che abitiamo.  
  
C'è però qualcosa che accomuna almeno tutte le classi che fungono da contenitori intelligenti per dati della più varia natura (ah, la templatizzazione... che grande trovata!) ossia la presenza in ciascuna di esse di un membro appartenente a un'altra classe templatizzata di assoluta rilevanza: la classe degli iteratori, fruibile con l'inclusione  
*# include <iterator>*  
sovente implicata da altre inclusioni attinenti.  
  
Un *iteratore* è una sorta di puntatore templatizzato che però è ANCHE un oggetto, ossia un'istanza di una certa classe, e NON SOLAMENTE una semplice variabile scalare, assimilabile, per quanto attiene al suo valore, a un volgare numero intero, come accade con i puntatori ordinari.   
  
Capite dove ci ha portato il nostro viaggio? In pratica al ricongiungimento tra puntatori e oggetti, come quando gli ardimentosi esploratori precursori di Amundsen cercavano il passaggio a nord-ovest per ricongiungere a occidente il nuovo mondo col vecchio, rimettendoci quasi tutti le penne dopo essersi lasciati alle spalle l'*ultima Thule*...  
Il destino dell'apprendista buon/a programmatore/trice C++ non è altrettanto gramo, anche perché egli/ella non sarà mai abbandonato/a dalla sua guida, e, giunto/a a questo punto, non potrà non apprezzare la disponibilità di *qualcosa* che:

* ha le potenzialità di un puntatore;
* è templatizzato, quindi può essere utilizzato con qualunque tipo;
* è un oggetto, quindi si porta appresso un ricco corredo di metodi membri, tra cui numerosi operatori.

(Graaaaan linguaggio...) Gli iteratori si possono raggruppare in 5 (cinque) grandi categorie; a partire dallo *standard* 2017 ne sarà introdotta una sesta, che si nominerà "categoria dei *Contiguous Iterators*", ma per l'appunto se ne parlerà, forse, al momento appropriato.  
  
Delle cinque presenti fin da oggi, quattro sono incluse una dentro l'altra e la quinta può avere intersezione non vuota con ciascuna delle altre quattro: se pensate ai disegnini che facevate alle elementari quando la maestra vi spiegava l'insiemistica, si tratta di quattro cerchi concentrici di raggio decrescente e di un quinto cerchio esterno che li intercetta tutti e quattro senza coprirne interamente nessuno.  
Senza timor di dubbio si deduce immediatamente che un iteratore che appartiene alla categoria rappresentata dal minore dei quattro cerchi concentrici appartiene certamente ANCHE alle categorie rappresentate dagli altri tre, mentre il viceversa non è detto che sia vero. Quanto alla quinta categoria temo di offendere la vostra logica mettendomi a disquisire su chi sia fuori e chi sia dentro.  
  
Diamo ora i nomi delle cinque categorie di iteratori attuali; già solo leggendoli si comprenderà parecchio sul loro significato. Gli iteratori appartenenti alla quinta categoria, quella rappresentata dal cerchio NON concentrico con gli altri quattro, sono detti *output iterators*, mentre quelli appartenenti alla categoria rappresentata dal maggiore dei quattro cerchi concentrici sono detti *input iterators*. Dato che si è già detto che l'intersezione di queste due categorie NON è vuota, gli iteratori che appartengono a entrambe sono chiamati *mutable iterators*.  
Addentrandosi all'interno della bolgia degli *input iterators*, superato il bordo del primo girone che vi si trova contenuto, si incontrano i cosiddetti *forward iterators*, dopo di che, nella terza bolgia, stanno i *bidirectional iterators* e infine, nella Giudecca...voglio dire... nel girone più interno, direttamente in bocca a Lucifero, stanno i massimi peccatori....intendo i *random access iterators*.  
  
La collocazione topologica dei futuri *contiguous iterators*, al momento presente non mi è noto quale finirà per essere.  
  
A quale categoria un iteratore appartenga dipende evidentemente da come viene istanziato, ossia da quale costruttore lo costruisce e, in definitiva, quando sono membri di una delle classi contenitore, da come questa lo gestisce. Ciò detto, il fatto di appartenere a una categoria piuttosto che a un'altra comporta con ogni evidenza una diversità di prestazioni da parte dell'iteratore di cui si tratta.  
  
Ecco dunque le caratteristiche peculiari di ogni categoria di iteratori, che sono contenute, alla fin fine, nel loro nome:

* gli *output iterators* e gli *input iterators* hanno prestazioni IDENTICHE, salvo il fatto, come dice il loro nome, di essere deputati i primi ad accedere agli oggetti *puntati* per modificarne il valore e i secondi SOLO a leggerne il valore e utilizzarlo. Possono essere incrementati di uno per farli accedere all'elemento successivo del contenitore cui appartengono, ma NON possono essere decrementati per accedere all'elemento precedente: in sostanza dispongono solo dell'*overload* dell'operatore *++*. Naturalmente i *mutable iterators* possono sia leggere sia scrivere...e possono anche appartenere alle categorie di cui si parla qui sotto;
* i *forward iterators*, siano o no *mutable iterators*, dànno in aggiunta la garanzia di poter essere usati senza tema di perdita di validità anche in algoritmi a passo multiplo: significa che offrono la possibilità di essere usati in un'operazione di assegnamento senza che questa renda l'iteratore non più valido;
* i *bidirectional iterators* hanno in aggiunta (indovinate!) gli *overload* degli operatori *--*; non credo di dover aggiungere altro;
* infine i *random access iterators*, oltre a riuscire a fare tutto quello che fanno anche gli iteratori delle altre categorie, sono capaci di accedere DIRETTAMENTE, ovvero senza dovere passare per forza attraverso gli elementi precedenti o successivi, a qualsiasi elemento del contenitore cui appartengono: in sostanza hanno gli *overload* degli operatori *[  ]*, *+=* e *-=*.

Ecco un breve esempio in cui si usano le classi templatizzate *map*, di cui viene usato un iteratore, e *string* per una gestione dei nomi senza preoccupazioni di spazio in memoria o di caratteri terminatori. Il programma è autocommentato.  
  
*# include <iostream>  
# include <map>  
# include <string>  
using namespace std;  
  
int main(  )  
{  
map<string, short int> \*   
voti\_presi\_all\_esame\_di\_programmazione = new map<string, short int>;  
  
/\*  
dichiarato e inizializzato un puntatore, dal nome parlante, all'istanza  
map<string, short int>   
della classe templatizzata map  
\*/  
  
while(true)  
{  
short int voto;  
string nome;  
cout << "digita un nome\n", getline(cin, nome);  
if(cin . eof(  )) break;  
cout << "digita un voto\n", cin >> voto;  
while(cin.get(  ) != '\n'); cin . clear(  );  
auto p = make\_pair(nome, voto);  
// |  
// |  
/\* > qui, grazie alla parola auto e alla funzione templatizzata make\_pair  
si crea un oggetto p di tipo "pair"  
pair non è altro che una tuple con due soli elementi ed è il  
tipico contenuto di una map  
\*/  
voti\_presi\_all\_esame\_di\_programmazione -> insert(p);  
// |  
// |  
// > che, infatti, viene inserito nel contenitore tramite il metodo insert  
}  
cout << "sono stati inseriti " << voti\_presi\_all\_esame\_di\_programmazione -> size(  )  
<< " studenti:\n";  
for(auto it1 = voti\_presi\_all\_esame\_di\_programmazione[0] . begin(  );  
it1 != voti\_presi\_all\_esame\_di\_programmazione[0] . end(  );  
++it1)  
// |  
// |  
/\* > questo ciclo esplora il contenitore tramite un iteratore: i metodi begin(  ) e end(  )  
(con evidenza) restituiscono "inizio" e "fine" del contenitore; tali "posizioni"  
sono valori plausibili per un iteratore, ed è per questo che il ciclo funziona.  
it1 è un "input iterator"  
\*/  
{auto p = \*it1;  
// |  
// |  
/\* > per quanto si è detto p è il contenuto del contenitore, quindi è una pair,  
di cui, sotto, vengono scritti i due elementi, identificati dalle variabili  
membri first e second  
\*/  
cout << p . first . data(  ) << " ha conseguito " << p . second << '\n';  
// comunque si siano forniti i dati, quando escono sono ORDINATI  
// GRAN LINGUAGGIO  
}  
cin . clear(  );  
string pappo;  
cout << "digita uno dei nomi che hai scritto\n", getline(cin, pappo);  
short int voto;  
cout   
<< "digita un voto che correggerà quello che "   
<< pappo << " ha conseguito\n", cin >> voto;  
while(cin.get(  ) != '\n'); cin . clear(  );  
voti\_presi\_all\_esame\_di\_programmazione[0][pappo] = voto;  
// |  
// |  
/\* > NON FATE CONFUSIONE: questo overload dell'operatore [  ] NON C'ENTRA  
NULLA con l'iteratore; non per questo it1 diventa un random access iterator...  
\*/  
for(auto it1 = voti\_presi\_all\_esame\_di\_programmazione[0] . begin(  );  
it1 != voti\_presi\_all\_esame\_di\_programmazione[0] . end(  );  
++it1)  
{  
auto p = \*it1;  
cout << p . first . data(  ) << " ha conseguito " << p . second << '\n';  
}  
while(true)  
{  
cout << "di chi vuoi sapere il voto ? ", getline(cin, pappo);  
if(cin .eof(  )) return 111;  
cout   
<< pappo << " ha conseguito "  
<< voti\_presi\_all\_esame\_di\_programmazione[0].at(pappo) << '\n';  
}  
}*Eseguite il programma, e quando vi verranno chiesti dei nomi digitate quello che vi pare. Quando vorrete smettere, al posto di un nuovo nome inserite il segnalatore di *End Of File* relativo alla tastiera, ossia simultaneamente i tasti *Ctrl* e **d**: questa è la ragione per cui, successivamente, è richiesta l'esecuzione di  
  
*cin.clear(  )*  
  
in modo da permettere la ricezione di ulteriori dati.  
  
Apprezzate il fatto che, qualora rispondiate alla richiesta di introdurre un nome già dato con la digitazione di un nome NUOVO, lungi dal produrre errore questo nuovo nome sarà semplicemente aggiunto al contenitore, e inserito nella giusta posizione ordinale.  
  
Notate anche che la funzione *getline* usata in questo programma NON È IL FAMILIARE METODO MEMBRO delle classi di *ios*, ma quella funzione dichiarata direttamente in **std** cui si era accennato nel canto sui metodi di *input*, e che era stata coperta dal silenzio (ricordate? RICORDATE!). Ora siete maturi per capire che si tratta di un'altra funzione templatizzata di cui l'oggetto gestore dell'*input stream* costituisce il primo argomento.  
  
Infine osservate come il buon programmatore si sia cautelato contro l'eventualità della presenza nell'*input stream* di un numero eccessivo di caratteri dovuti a troppe e incontrollate pigiature del tasto Invio, che potrebbero desincronizzare la lettura di dati misti come sono stringhe e numeri.  
  
Questo canto termina con una rassegna DETTAGLIATA delle caratteristiche irrinunciabili che deve possedere una classe affinché una sua istanza possa fregiarsi del titolo di iteratore: in questo modo il buon programmatore, oltre a utilizzare con cognizione di causa quelli previsti dallo *standard* nei diversi contenitori, potrà perfino riuscire a definirne di PROPRI.  
  
Tanto per cominciare esistono, fruibili con l'inclusione dell'*header* nomato appunto *iterator*, le seguenti due classi templatizzate che costituiscono il fondamento di QUALSIASI iteratore:

* *template  
  <class C, class T, class D = std::ptrdiff\_t, class P = T\*, class R = T&>  
  struct iterator;*
* *template <class I> struct iterator\_traits;*

Queste due *template struct* sono completamente vuote, o meglio non hanno altri membri che non siano delle *typedef*inizioni di tipi utili a istanziare iteratori VERI: ne segue che quando se ne voglia creare uno sarà opportuno che la sua classe sia resa erede della prima di queste e sfrutti adeguatamente le informazioni fornite dalla seconda.  
  
**Esempio inutile:**  
  
*# include <iterator>  
  
template <class C, class T> struct iteratore\_mio : std::iterator<C, T>  
{typename std::iterator\_traits<iteratore\_mio>::difference\_type diff;};  
  
int main(  ) {iteratore\_mio<std::input\_iterator\_tag, int> iter;}*   
  
L'esempio, pur nella sua primordiale inutilità, si compila e si esegue senza alcun errore, e vi presenta la prima istanziazione di un oggetto, *iter*, dichiarato in *main*, che è già QUASI un iteratore, e per giunta incontestabilmente creato sui due pie'.  
  
Per rendere ragione del *"QUASI"*, ossia di quel che manca a *iter* per essere a pieno titolo un iteratore, bisogna partire da quello che *iter* già ha per provare a esserlo: nessuno potrà non essersi accorto che la *struct iterator*, da cui la classe *iteratore\_mio* eredita, ha solo DUE parametri di templatizzazione liberi, gli altri essendo tutti dotati di valori *standard*, e per giunta con relazioni alquanto evidenti del penultimo e dell'ultimo con il secondo.  
  
Gli stessi due parametri di templatizzazione sono istanziati in *iteratore\_mio* nel modo indicato: il primo tramite un tipo *typedef*inito nel *namespace std* e il secondo come il familiare tipo nativo *int*. Dovrebbe risultare abbastanza intuitivo che, accanto al tipo *std::input\_iterator\_tag* vi debbano essere anche

* *std::output\_iterator\_tag*
* *std::forward\_iterator\_tag*
* *std::bidirectional\_iterator\_tag*
* *std::random\_access\_iterator\_tag*
* *std::contiguous\_iterator\_tag*

e che ciascuno di questi tipi debba essere usato per istanziare un iteratore della categoria che i nomi dei tipi richiamano in modo palese.  
  
Il fatto che *iter* possegga una variabile membro chiamata *diff* non è istruttivo di per sé: lo è molto di più il tipo che le è stato attribuito e che viene definito, in modo evidente, dalla *template struct iterator\_traits* istanziata col SUO tipo templatizzato che è PROPRIO la NOSTRA classe; accanto a quel tipo, che, per come è stata istanziata la nostra classe, CONSERVA il valore *standard* del parametro di templatizzazione *D*, ossia il terzo, *iterator\_traits* ne *typedef*inisce diversi altri, nomati:

* *value\_type*
* *pointer*
* *reference*
* *iterator\_category*

Ognuno di questi ha un significato chiaro rispetto all'istanziazione di *iteratore\_mio* utilizzata: in particolare *value\_type* finisce per coincidere con l'istanziazione corrente del parametro di templatizzazione *T*, il secondo, così che tanto *pointer* quanto *reference* e *iterator\_category* assumono il senso che ognuno dovrebbe intuire. Quanto a *difference\_type* si tratta di un tipo idoneo a dichiarare variabili che abbiano appunto il valore che compete alla "distanza", con segno, tra iteratori omologhi e quindi, con verosimiglianza, finirà con l'essere una versione *signed* di *pointer*: la cosa bella è che non occorre preoccuparsene fintanto che si usano i nomi qui introdotti per dichiarare variabili membro, come *diff*. Ovviamente NESSUNO impedisce al buon programmatore di *typedef*inire anche le *typedef*inizioni *standard*, sì da renderle più brevi e intuitive.  
  
È giunto il momento di formulare i REQUISITI MINIMI che una classe DEVE POSSEDERE per potersi fregiare del titolo di plausibile iteratore, e si tratta di:

* la classe DEVE essere copiabile, ossia avere un costruttore di copia accessibile;
* la classe DEVE essere assegnabile, ossia avere un *copy assign operator=* accessibile;
* la classe DEVE essere distruggibile, ossia avere un distruttore accessibile;
* la classe DEVE essere interscambiabile, ossia due sue istanze DEVONO poter essere trasmesse alla funzione *swap* e ottenerne l'effetto;
* la classe DEVE essere dereferenziabile, ossia: data una sua istanza *IST* DEVE aver senso l'espressione *\*IST*, pur senza alcuna richiesta specifica sul tipo di tale espressione;
* la classe DEVE essere incrementabile, ossia: data una sua istanza *IST* DEVE aver senso l'espressione *++IST* e il tipo di questa espressione DEVE essere un riferimento sinistro alla classe stessa di *IST*.

Anche un esame superficiale di *iteratore\_mio* consente di capire che le ragioni del *"QUASI"* vanno identificate nel mancato soddisfacimento dei SOLI ULTIMI DUE PUNTI. Pertanto la seguente variante del codice precedente, in cui sono messe in luce tutte le adesioni ai requisiti prescritti, costituisce la PRIMA APPLICAZIONE, tuttavia ancora del tutto INUTILE, di una classe che HA PIENO DIRITTO DI RIVENDICARE per sé la qualifica di iteratore:  
  
*# include <iostream>  
# include <iterator>  
  
template <class C, class T> struct iteratore\_mio : std::iterator<C, T>  
{typename std::iterator\_traits<iteratore\_mio>::difference\_type diff;  
double operator\*(  ) {return 1.0;}  
iteratore\_mio& operator++(  ) {diff = -14; return \*this;}};  
  
template <class X> void funza(X i)  
{X j; j = i;} // copy assignable  
  
std::ostream& operator<<  
(std::ostream& o, iteratore\_mio<std::input\_iterator\_tag, int> &i)  
{return o << i.diff;}  
  
int main(  )  
{iteratore\_mio<std::input\_iterator\_tag, int> iter, iter2;  
funza(iter2), // copy constructible  
std::cout << ++iter << std::endl; // incrementabile  
std::swap(iter, iter2); // swappabile  
iter.~iteratore\_mio<std::input\_iterator\_tag, int>(  );//distruggibile  
std::cout << \*iter2 << std::endl; // dereferenziabile  
std::cout << iter2 << std::endl; // swap ha agito correttamente  
}*   
  
Se qualcuno si chiedesse perché questa applicazione sia stata gabellata come INUTILE ... significa che ha gli occhi della mente foderati di pelle di salame: la classe *iteratore\_mio* così scritta è sì un iteratore, ma NON un *input iterator*, che era poi la ragione che l'aveva ispirata al buon programmatore, NÉ un iteratore di qualunque altra categoria. A ben vedere, e quindi **RIVEDETE**, i requisiti minimi per avere una classe iteratore possono essere soddisfatti DA DOVIZIE DI CLASSI, senza NEMMENO BISOGNO che siano templatizzate ed eredi di *iterator*: devono solo avere, in un ambito ANCHE VUOTO, un membro *operator\** unario e uno *operator++* prefisso; ecco quindi che cosa bisogna aggiungere, SENZA DIMENTICARE quanto già richiesto, per ottenere un *input iterator* VERO e PROPRIO:

* il tipo dell'espressione *\*IST*, ove *IST*, al solito, è un'istanza della classe, non può essere qualsiasi ma DEVE essere un riferimento sinistro al tipo individuato da *value\_type*; e già questo rende ragione della templatizzazione e dell'ereditarietà. Nel caso concreto di *iteratore\_mio* il tipo restituito da *operator\** DEVE finire per essere *int&*, E NON AFFATTO *double*.
* la classe DEVE essere paragonabile, ossia dare un senso, e QUELLO INTUIBILE, a espressioni di confronto tra due sue istanze *IST1* e *IST2*; significa che devono esistere, col significato intuibile, gli *operator==* e *operator!=*, ed entrambi devono restituire un tipo convertibile a *bool*.
* come corollario dei due punti precedenti, se *IST1==IST2* ha valore *true*, allora *\*IST1* e *\*IST2* DEVONO ESSERE EQUIVALENTI.
* la classe DEVE AVERE un membro che esegua l'*overload* dell'operatore freccia (*->*) e questo *overload* deve consentire l'accesso a un membro *membro* della classe tramite l'espressione *IST->membro*, come se fosse *(\*ptIST).membro*, in cui *ptIST* sia un puntatore a un'istanza della classe.
* la classe DEVE avere ANCHE un *operator++* suffisso, e questo deve svolgere il compito intuibile espresso dalle seguenti tre istruzioni che, in pratica, ne costituiscono l'ambito:  
  *value\_type val = \*\*this;  
  ++\*this;  
  return val;*   
  ne segue che il tipo restituito sia *value\_type* invece che un riferimento sinistro alla classe, ma che comunque l'effetto collaterale sull'oggetto proprietario è il medesimo, di modo che le due seguenti espressioni, in cui l'effetto collaterale è l'unica cosa che conta,  
  *(void)++IST;  
  (void)IST++;*siano da considerare perfettamente equivalenti.

Ecco dunque la classe *iteratore\_mio* DEFINITIVAMENTE TRASFORMATA in un EFFETTIVO *input iterator* fabbricato in casa, con tanto di *main* per usarlo; i commenti che accompagnano il codice sono stati scritti per evidenziare ciò che, secondo l'opinione giusta di chi scrive, meritava qualche sottolineatura:  
  
*# include <iostream>  
# include <iterator>  
# include <vector>  
  
// un input iterator templatizzato, fatto in casa:  
template <class C, class T, class W = std::iterator\_traits<T\*>>  
struct iteratore\_mio : std::iterator<C, T>  
{  
// un filotto di typedefinizioni, giusto per accorciare le dichiarazioni:  
typedef typename std::iterator\_traits<iteratore\_mio>::difference\_type Diff;  
typedef typename std::iterator\_traits<iteratore\_mio>::value\_type Val;  
typedef typename std::iterator\_traits<iteratore\_mio>::reference Ref;  
typedef typename std::iterator\_traits<iteratore\_mio>::pointer Ptr;  
typedef typename std::iterator\_traits<iteratore\_mio>::iterator\_category It;  
// alcune variabili membro, pubbliche perché siamo in una struct:  
Diff diff;  
Val val;  
Ptr ptr;  
size\_t curr, estensione;  
bool valido;  
W inizio;  
T tt;  
// taluni dei membri sono a piacere; altri sono necessari.  
// Comunque se ne possono aggiungere altri, secondo il proprio talento  
//  
// l'operatore freccia, RICHIESTO dallo standard per poter essere un iteratore:  
iteratore\_mio\* operator->(  ) {return this;}  
//  
// gli operatori RICHIESTI per == e !=  
bool operator==(iteratore\_mio secondo)  
{return val == secondo.val;}  
bool operator!=(iteratore\_mio secondo)  
{return ! (\*this==secondo);}  
//  
// un costruttore parametrico, utile per la SECONDA istanziazione della  
// classe operata da main. In essa il terzo parametro di templatizzazione  
// viene sovrascritto; se si prevedono altre istanziazioni è  
// presumibile che si possano/debbano introdurre altri costruttori e  
// altre variabili membro:  
iteratore\_mio(T t) : tt(t), curr(0), valido(true)  
{inizio = t.begin(  );  
estensione = std::distance(t.begin(  ), t.end(  ));  
val = static\_cast<Val>(\*inizio); ptr = &val;}  
//  
//  
// il costruttore default, NECESSARIO per la presenza del precedente  
// costruttore e UTILIZZATO nella PRIMA istanziazione della classe in main.  
// Nessuna cautela è stata posta contro l'eventuale utilizzo di questo  
// costruttore senza trasmettergli alcun parametro:  
iteratore\_mio  
(Ptr p = nullptr,  
typename W::pointer it = nullptr,  
typename W::pointer ot = nullptr)  
: ptr(p), curr(0), valido(true)  
{val = p[0];  
estensione = std::distance(it, ot);}  
//  
//  
// alcuni metodi membro NON indispensabili; altri metodi da aggiungere  
// secondo FANTASIA:  
Diff distanza(iteratore\_mio secondo)  
{return diff = (Diff)(ptr+curr) - (Diff)(secondo.ptr+secondo.curr);}  
Diff distanza(  ) {return curr;}  
//  
//  
// l'operatore \* unario, RICHIESTO dallo standard:  
Ref& operator\*(  ) {return val = ptr[curr];}  
//  
//  
// gli operatori ++ suffisso e prefisso, RICHIESTI dallo standard  
Val operator++(int) {Val x = \*\*this; ++\*this; return x;}  
iteratore\_mio& operator++(  )  
{++curr;  
iteratore\_mio dummy(\*this);  
if(distanza(  ) >= estensione)  
std::cerr << "iteratore invalidato\n", valido = false;  
return \*this;}};  
// FINITA la definizione di iteratore\_mio  
  
  
// una funzione templatizzata, istanziabile SOLO con istanze di iteratore\_mio  
// altrimenti si rischia forte, secondo le regole della template substitution  
// non le va fatto fare niente altro, visto come istanzia la variabile locale j.  
template <class X> void funza(X i)  
{X j(i->ptr); j = i;}  
  
  
//  
// DUE overload dell'operator<<, uno per ciascuna istanza di iteratore\_mio  
// in main: si è preferita questa via piuttosto che la templatizzazione  
// ANCHE di operator<<, che avrebbe richiesto una diversa e maggiore attenzione  
// sia nella sua definizione sia nel suo uso:  
std::ostream& operator<<  
(std::ostream& o, iteratore\_mio<std::input\_iterator\_tag, int> &i)  
{return o << i->val;}  
std::ostream& operator<<  
(std::ostream& o, iteratore\_mio<std::input\_iterator\_tag, std::vector<int>, std::vector<int>::iterator> &i)  
{i->inizio = i->tt.begin(  );  
return o << \*(i->inizio + i->curr);}  
  
  
// ...... main ........  
int main(  )  
{  
// due array di int, di uguale sizeof:  
int x[  ]{1, 2, 3, 4, 5}, y[  ]{10, 20, 30, 40, 50};  
//  
//  
// creazione di due oggetti della classe iteratore\_mio, secondo  
// l'istanziazione della medesima con terzo parametro standard:  
// qui C diventa std::input\_iterator\_tag  
// e T diventa int  
// per questa ragione possono essere trasmessi i due array come primo  
// parametro e gli esiti delle funzioni begin e end su ciascuno di essi  
// come parametri successivi del costruttore di default:  
iteratore\_mio<std::input\_iterator\_tag, int>  
iter(x, std::begin(x), std::end(x)), iter2(y, std::begin(y), std::end(y));  
//  
//  
//  
// la richiesta di eseguire funza è accettata con  
// X dedotto come il tipo di iter2, perché si tratta  
// di un tipo copy constructible:  
funza(iter2);  
//  
// anche la creazione di un ulteriore oggetto dello stesso tipo  
// va a buon fine per la STESSA ragione; iter3 è una COPIA di iter  
iteratore\_mio<std::input\_iterator\_tag, int> iter3(iter);  
std::cout << "ecco i valori di alcune variabili membro di due oggetti COPIATI\n"  
<< iter->val << ' ' << iter->curr << '\n'  
<< iter3->val << ' ' << iter3->curr << '\n';  
//  
//  
//  
// ora si osservi il valore restituito dal metodo distanza di iter3  
// quando gli si trasmetta la copia di iter INCREMENTATO:  
std::cout << iter3 -> distanza(++iter) << '\n';  
//  
//  
// ecco il valore restituito da operator\* di iter, ULTERIORMENTE  
// incrementato:  
std::cout << \*++iter << std::endl;  
//  
//  
// ora si scambiano tra loro iter e iter2, perché la classe  
// DEVE ESSERE swappabile  
std::swap(iter, iter2);  
//  
// dopo di che si invoca VOLONTARIAMENTE il distruttore di iter,  
// perché la classe DEVE ESSERE distruggibile:  
iter.~iteratore\_mio<std::input\_iterator\_tag, int>(  );  
//  
// ma i valori di iter sono in iter2, mercè il precedente  
// intervento di swap; infatti ecco che cosa restituisce  
// l'operator \* di iter2:  
std::cout << \*iter2 << std::endl;  
//  
//  
// che può essere ottenuto anche attraverso operator<<:  
std::cout << iter2 << std::endl << "\n\n\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n";  
//  
//  
// mercè il seguente ciclo si può apprezzare che la  
// classe è in grado di accorgersi quando non può  
// più essere incrementata oltre. Al posto della variabile  
// membro valido si sarebbe potuto definire un metodo membro end(  )  
// e anche un metodo membro begin(  ) con ovvii significati e che  
// avrebbero reso iteratore\_mio ancora più simile agli  
// iteratori standard del linguaggio:  
do std::cout << \*iter2 << std::endl; while(++iter2, iter2->valido);  
//  
//  
// ora si crea e si riempie col contenuto dell'array x un oggetto  
// della classe STL vector<int>; si rammenta che questa HA i suoi  
// iteratori GIÀ DEFINITI:  
std::vector<int> vettore;  
for(auto i: x) vettore.push\_back(i);  
//  
//  
// ma anche le seguenti appropriate istanze di iteratore\_mio  
// si comportano correttamente a proposito dell'oggetto vettore  
// trasmesso come parametro all'appropriato costruttore:  
iteratore\_mio  
<std::input\_iterator\_tag, std::vector<int>, std::vector<int>::iterator>  
iter4(vettore);  
iteratore\_mio  
<std::input\_iterator\_tag, std::vector<int>, std::vector<int>::iterator>  
iter5(vettore);  
iteratore\_mio  
<std::input\_iterator\_tag, std::vector<int>, std::vector<int>::iterator>  
iter6(iter5);  
//  
//  
// tanto per cominciare sono riconosciute come UGUALI:  
if(iter4 == iter5) std :: cout << "Questi due iteratore\_mio sono uguali\n";  
//  
//  
//  
// e anche gli operator++ e operator\* (quest'ultimo opportunamente  
// travestito entro operator<<) forniscono i risultati aspettati:  
do std::cout << iter4 << std::endl; while(++iter4, iter4->valido);  
//  
// e quant'altro .....:  
std::cout << "Ora stiamo per ripetere il controllo di uguaglianza\n" << std::flush;  
if(iter4 == iter5) std :: cout << "Questi due iteratore\_mio sono uguali\n";  
std::cout << "... e che ne è di iter6 ?\n";  
if(iter4 == iter6) std::cout<<"Anche questi sono uguali\n";  
if(iter5 == iter6) std::cout<<"proprietà transitiva...\n";  
if(!(iter5 != iter6)) std::cout << "...evidentemente...\n";  
}*  
  
Se poi si vorrà promuovere *iteratore\_mio* alla categoria dei *forward iterators*, i quali restano comunque ANCHE degli *input iterators*, bisognerà tener conto delle caratteristiche **aggiuntive**, e pertanto maggiormente restrittive, che lo *standard* impone. Esse sono:

* la classe del candidato *forward iterator* DEVE ANCHE essere *default constructible*; per quanto riguarda *iteratore\_mio* questo significa che il costruttore *default* non può più permettersi di inizializzare variabili membro lasciando passare in cavalleria il fatto che abbia o no ricevuto gli argomenti con cui le inizializza.
* la classe del candidato *forward iterator* DEVE GARANTIRE il cosiddetto *multipass*, ossia:
  + se *iter1* e *iter2* sono due sue istanze e l'espressione *iter1 == iter2* ha valore *true*, ALLORA *\*iter1* e *\*iter2* DEVONO restituire riferimenti sinistri ALLO STESSO OGGETTO, oppure DEVONO ESSERE ENTRAMBI non (più) dereferenziabili; allo stato attuale questo non avviene, IN GENERALE, con *iteratore\_mio*.
  + se l'operatore ++ agisce su una copia dell'iteratore, ciò NON DEVE INFLUENZARE il risultato dell'operatore di dereferenza sull'iteratore originale.
  + dall'eventuale valore *true* dell'espressione *iter1 == iter2* DEVE SEMPRE SEGUIRE VALORE *true* ANCHE dell'espressione *++iter1 == ++iter2*
  + qualora si volesse rendere *mutable*, vale a dire ANCHE *output iterator*, il candidato *forward iterator*, l'operatore di assegnamento che lo abbia come operando sinistro NON DEVE INVALIDARLO.
* Il tipo contemplato da *std::iterator\_traits<candidato\_iteratore>::reference*, in cui, evidentemente, *candidato\_iteratore* è il nome fittizio della classe che definisce il candidato *forward iterator*, DEVE COINCIDERE ESATTAMENTE con *const T&* se l'iteratore non deve essere *mutable* e con *T&* nel caso contrario. Qui, evidentemente, *T* è il secondo tipo *template* nell'istanziazione della classe.
* Gli operatori == e != devono essere definiti per TUTTI gli iteratori relativi allo stesso contenitore di oggetti, anche fossero inizializzati per valore, ossia inizializzati a "dopo la fine" del contenitore: il valore restituito dal metodo *end(  )*, tanto per capirsi.

Tutto ciò elencato e digerito, ecco una possibile maniera di trasformare efficacemente *iteratore\_mio* in un *forward iterator*; ancora una volta i commenti inseriti nel codice, assieme a un confronto diretto con la versione precedente, dovranno aiutare a capire i punti cruciali della metamorfosi. Ovviamente, nel caso si pretendessero altre istanziazioni, si dovrà tenere debito conto delle aggiunte e/o dei cambiamenti imposti dalla *template substitution*, magari introducendo *overload* templatizzati per gli operatori e, perché no?, addirittura degli *SFINAE* inseriti al punto giusto: a questo punto della Commedia dovreste essere in grado anche di andare a leggere e comprendere la pagina appositamente scritta...  
Si presti attenzione agli *escamotages* posti in essere anche nella versione che state per leggere, proprio al fine di parare le stoccate della *template substitution* per due istanziazioni COSÌ LONTANE come sono quelle realizzate in *main*, ossia una volta con la classe *vector<int>* e l'altra con un TIPO **VUOTO**.  
  
*# include <iostream>  
# include <iterator>  
# include <vector>  
# include <cstdlib>  
# include <type\_traits>  
using namespace std;  
  
template <class C, class T, class W = std::iterator\_traits<T\*>>  
struct iteratore\_mio : std::iterator<C, T>  
{  
typedef typename std::iterator\_traits<iteratore\_mio>::difference\_type Diff;  
typedef typename std::iterator\_traits<iteratore\_mio>::value\_type Val;  
// si osservi, qui sotto, l'aggiunta di const;  
// non si sta definendo un output iterator  
typedef typename std::iterator\_traits<iteratore\_mio>::reference const Ref;  
typedef typename std::iterator\_traits<iteratore\_mio>::pointer Ptr;  
typedef typename std::iterator\_traits<iteratore\_mio>::iterator\_category It;  
Diff diff;  
Val val;  
Ptr ptr;  
Ref ref = val, reff = tt;  
size\_t curr, estensione, corr;  
bool valido;  
W \* inizio, itt;  
T tt;  
iteratore\_mio\* operator->(  ) {return this;}  
  
bool operator==(iteratore\_mio &secondo)  
{bool b;  
if(ptr == nullptr && secondo.ptr == nullptr) return valido == secondo.valido;  
else if(ptr == nullptr || secondo.ptr == nullptr) return false;  
else  
{b = \*ptr == \*secondo.ptr;  
if(!is\_class<T>::value) return b;  
return b && corr == secondo.corr;}}  
  
bool operator!=(iteratore\_mio &secondo)  
{return ! (\*this==secondo);}  
  
iteratore\_mio(T &t) : tt(t), curr(0), corr(0), valido(true)  
{inizio = new W(t.begin(  ));  
estensione = std::distance(t.begin(  ), t.end(  ));  
val = static\_cast<Val>(\*\*inizio); ptr = &val;}  
  
iteratore\_mio  
(Ptr p = nullptr,  
typename W::pointer it = nullptr,  
typename W::pointer ot = nullptr)  
: ptr(p), curr(0),  
corr(random(  )\*static\_cast<size\_t>(!is\_class<T>::value)),  
valido(false)  
{if(p) val = p[0], valido = true;  
inizio = reinterpret\_cast<W\*>(random(  ));  
estensione = std::distance(it, ot);}  
  
  
// per un forward iterator occorre un costruttore di  
// copie fatto con attenzione, a maggior ragione se si  
// tratta di un iteratore templatizzato come iteratore\_mio  
iteratore\_mio(const iteratore\_mio &originale)  
{valido = originale.valido;  
curr = originale.curr;  
corr = originale.corr;  
val = originale.val;  
diff = originale.diff;  
ptr = originale.ptr;  
if(originale.ptr && !is\_class<T>::value) \*ptr = \*originale.ptr, ref = val;  
estensione = originale.estensione;  
tt = originale.tt;  
itt = originale.itt;  
inizio = new W;  
\*inizio = \*originale.inizio;  
reff = tt;}  
  
  
// funzioni suppletive dell'azione di un costruttore  
// default effettivo (eseguito SENZA argomenti)  
void set(Ptr const p, size\_t u)  
{ptr = p, estensione = u,  
val = p[0], curr = 0,  
corr = random(  ), valido = true;}  
void set(T &v)  
{tt = v, estensione = v.end( ) - v.begin( ), ptr = &v;  
inizio = new W, \*inizio = v.begin(  );  
if(!is\_class<T>::value) corr = random(  );  
valido = true, itt = v.begin(  );}  
  
Diff distanza(iteratore\_mio secondo)  
{return diff = (Diff)(ptr+curr) - (Diff)(secondo.ptr+secondo.curr);}  
  
Diff distanza(  )  
{if(!is\_class<T>::value) return curr; return corr;}  
  
Ref& operator\*(  ) // Ref ha in sé la qualifica const  
{if(!is\_class<T>::value) return ref = val; return reff = tt;}  
  
Val operator++(int) {Val x = \*\*this; ++\*this; return x;}  
  
iteratore\_mio& operator++(  )  
{if(!is\_class<T>::value) ++curr, val = ptr[curr];  
else ++corr;  
if(distanza(  ) >= estensione)  
std::cerr << "iteratore invalidato\n", valido = false;  
return \*this;}};  
  
  
std::ostream& operator<<  
(std::ostream& o, iteratore\_mio<std::forward\_iterator\_tag, int> &i)  
{if(!i->valido) return o;  
return o << i->ref;}  
std::ostream& operator<<  
(std::ostream& o, iteratore\_mio<std::forward\_iterator\_tag,  
std::vector<int>, std::vector<int>::iterator> &i)  
{if(!i->valido) return o;  
\*i->inizio = i->tt.begin(  );  
return o << \*(\*i->inizio + i->corr);}  
  
template <typename T> void funza(T t)  
{cout << "nella template funza " << ++t << endl;}  
  
int main(  )  
{iteratore\_mio<forward\_iterator\_tag, int> iter1;  
iteratore\_mio<forward\_iterator\_tag, int> iter2(iter1);  
// ci si attende che iter1 e iter2 siano considerati uguali  
// (per copia di un iteratore inizializzato default)  
if(iter1 == iter2) cout << "iter1 e iter2 UGUALI\n";  
vector<int> vettorex, vettorey;  
iteratore\_mio<forward\_iterator\_tag, vector<int>, vector<int>::iterator> iter3, iter4;  
// ci si attende che ANCHE iter3 e iter4 siano considerati uguali  
// (per inizializzazione default di ENTRAMBI)  
if(iter3 == iter4) cout << "iter3 e iter4 UGUALI\n";  
int x[  ]{1, 2, 3, 4, 5}, y[  ]{-1, -2, -3, -4, -5};  
for(auto i : x) vettorex.push\_back(i);  
for(auto i : y) vettorey.push\_back(i);  
// attribuzione dei dati su cui iterare a iter1 e iter2:  
// trattandosi di due DISTINTI array i due iteratori si differenziano  
// uno dall'altro  
iter1->set(&\*x, end(x)-begin(x)), iter2->set(&\*y, end(y)-begin(y)),  
// attribuzione dei dati su cui iterare a iter3 e iter4:  
// trattandosi di due DISTINTI vector<int> i due iteratori  
// si differenziano uno dall'altro  
iter3->set(vettorex), iter4->set(vettorey);  
// ci si attende quindi che iter1 e iter2 NON SIANO   
// PIÙ UGUALI e che quindi il cout non sia eseguito  
if(iter1 == iter2) cout << "UGUALI 1-2\n"<<flush;  
// altrettanto ci si aspetta dal controllo di uguaglianza  
// fra iter3 e iter4  
if(iter3 == iter4) cout << "UGUALI 3-4\n"<<flush;  
// viceversa devono dare esito positivo i corrispettivi  
// controlli di DISUGUAGLIANZA  
if(iter1 != iter2) cout << "iter1 e iter2 DIVERSI\n"<<flush;  
if(iter3 != iter4) cout << "iter3 e iter4 DIVERSI\n"<<flush;  
// a conferma di quanto appena visto, ecco che cosa fornisce  
// l'operator\* unario applicato a ogni iteratore\_mio  
// (per la seconda istanza si ricorda che vi è un  
// overload dedicato di operator <<)  
cout << \*iter1 << ' ' << \*iter2 << '\n'<<flush;  
cout << iter3 << ' ' << iter4 << '\n'<<flush;  
// viene ora incrementato iter3 e ne viene riscritto il valore  
cout << ++iter3 << '\n'<<flush;  
// ...e iter3 permane DIVERSO da iter4  
if(iter3 == iter4) cout << "iter3 e iter4 UGUALI\n"<<flush;  
if(iter3 != iter4) cout << "iter3 e iter4 DIVERSI\n"<<flush;  
// viene ora creato un iteratore\_mio iter5, COPIA di iter3  
// (che era stato già incrementato rispetto alla sua creazione)  
iteratore\_mio<forward\_iterator\_tag, vector<int>, vector<int>::iterator>  
iter5(iter3);  
// iter3 e iter5 dovrebbero risultare uguali...  
if(iter3 == iter5) cout << "iter3 e iter5 UGUALI\n";  
// infatti, estraendo e visualizzando da ENTRAMBI  
// il valore del LORO iteratore predefinito begin(  )...  
int u3 = \*(\*iter3).begin(  ), u5 = \*(\*iter5).begin(  );  
cout << "u3 = " << u3 << "; u5 = " << u5 << '\n';  
// ora si incrementano sia iter3 sia iter5  
++iter3, ++iter5;  
// e se ne estraggono i valori correnti tramite la classe iteratore\_mio  
u3 = \*iter3->itt+iter3->corr; u5 = \*iter5->itt+iter5->corr;  
cout << "u3 = " << u3 << "; u5 = " << u5 << '\n';  
// NON solo u3 e u5 risultano uguali, ma anche e TUTTORA i due  
// iteratori, in ossequio a quanto richiesto a un forward iterator.  
// infatti...  
if(iter3 == iter5) cout << "iter3 e iter5 tuttora UGUALI\n";  
// ora iter3 viene trasmesso a funza, che lo riceve per copia e lo  
// incrementa localmente (vedere il valore scritto)  
funza(iter3);  
// ma nell'ambito di main iter3 resta uguale a iter5...  
if(iter3 == iter5) cout << "iter3 e iter5 tuttora e sempre UGUALI\n";  
// e il valore estratto da iter3...  
int u3\_ancora = \*iter3->itt+iter3->corr;  
// è lo stesso di prima, sempre ossequiando quanto  
// richiesto a un forward iterator  
cout << "u3(ancora) = " << u3\_ancora << '\n';  
while(iter2->valido) funza(iter2), ++iter2;  
cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n";  
while(iter3->valido) cout << ++iter3 << ' ';  
cout << endl;  
if(iter3 != iter5) cout << "iter3 e iter5 ora DIVERSI\n"; }*  
  
Portata, S.E.O., a compimento la missione più difficile, ossia il superamento della prima cornice infernale e l'ingresso nella bolgia dei *forward iterators* partendo dal rassicurante limbo degli *input iterators*, il resto della discesa è decisamente più semplice e la scrittura di un *bidirectional iterator* e, successivamente, di un *random access iterator* vengono lasciate come esercizio a chi legge. Basta sapere e tener presente che un *bidirectional iterator* non è altro che un *forward iterator* per cui sono definiti ANCHE gli *operator--*, sia prefissi sia suffissi, con una semantica e con vincoli IDENTICI a quelli posti per gli *operator++* di un *forward iterator*.  
Quanto al *random access iterator*, non si tratta che di un *bidirectional iterator*, testé discusso, cui si aggiungano le definizioni di:

* *operator +=* (membro, con operando destro intero *n*: deve equivalere, con gli stessi vincoli e semantica, a un ciclo percorso *abs(n)* volte e in cui, ogni volta, si applichi *operator++* o *operator--* secondo il segno di *n*)
* *operator -=* (membro, con operando destro intero *n*: speculare al precedente)
* *operator +* (binario, membro, con operando destro intero; equivale alla creazione di un iteratore della stessa classe, e di categoria *rvalue*, inizializzato con l'operando sinistro cui sia stato applicato *operator+=* con lo stesso operando destro)
* *operator -* (binario, membro, con operando destro intero; equivale alla creazione di un iteratore della stessa classe, e di categoria *rvalue*, inizializzato con l'operando sinistro cui sia stato applicato *operator-=* con lo stesso operando destro)
* *operator +* (binario, NON membro, con operando sinistro intero; equivale alla commutazione con l'omonimo operatore membro)
* *operator -* (binario, NON membro, con operando sinistro intero; equivale alla commutazione con l'omonimo operatore membro)
* *operator -* (binario, membro, con operando destro un altro iteratore della stessa classe; equivale alla soluzione dell'equazione *iter2 = iter1 + n* per l'incognita *n*, ossia restituisce, come *iterator\_traits<>::difference\_type*, l'intero *n* che dovrebbe essere trasmesso alla *operator-* membro di *iter2* per produrre un iteratore uguale a *iter1*)
* *operator[  ]* (binario, membro, con operando destro intero *n*; deve equivalere alla dereferenza dell'iteratore *rvalue* prodotto dalla *operator+* membro, ossia *iter1[n]* deve coincidere con *\*(iter1+n)*)
* TUTTI gli operatori binari di confronto con argomento destro un altro iteratore della stessa classe, con la convenzione che l'espressione  
  *iter1 < iter2*   
  DEVE:
  + avere un valore convertibile in *bool*;
  + se tale valore è *true* devono valere TUTTE le seguenti implicazioni:
    - *iter2 - iter1* restituisce un valore strettamente positivo
    - *! (iter1 < iter1)* deve essere pure *true*, e lo stesso con *iter2*
    - *! (iter2 < iter1)* deve essere pure *true*
    - se *iter2 < iter3* è pure *true* ALLORA DEVE esserlo anche *iter1 < iter3*
    - tanto *iter2 < iter1* quanto *iter2 == iter1* DEVONO ENTRAMBE valere *false*

Discorsi perfettamente analoghi, previa inversione di tutte le disuguaglianze, varranno evidentemente nel caso che sia *true* il valore dell'espressione  
*iter1 > iter2*.  
Quanto alle espressioni che coinvolgono l'*operator<=*, e, specularmente, l'*operator>=*, basti dire che l'espressione  
*iter1 <= iter2*  
DEVE AVERE LO STESSO valore, ed essere totalmente equivalente sotto ogni altro aspetto, alla negazione booleana di *iter1 > iter2*.

Altri argomenti da affrontare in questo canto riguardano la categoria degli *output iterators*: come già si è detto ripetutamente ciascuna delle categorie fin qui presentate si può fregiare dell'appellativo di *mutable iterator* qualora le accada di soddisfare anche le richieste poste dallo *standard* per un *output iterator*. Quali sono dunque queste richieste? Dovrebbe essere facile arguire che non debbano essere in contrasto con quelle poste a un *input iterator* ed estremamente più deboli di quelle cui devono soddisfare le altre categorie, altrimenti come si potrebbe farle soddisfare da tutte? A questo proposito va sottolineato che la PIENA ADERENZA ai requisiti richiesti dallo *standard* serve solo per poter attribuire fondatamente la categoria di appartenenza di un iteratore: non significa che una classe cui mancasse qualcuna delle proprietà dovute sia da buttare come inutile. Ad esempio, se dalla definizione di un potenziale *random access iterator* si rimovesse l'*overload* dell'*operator-* che restituisce il tipo *difference\_type*, a rigor di termini NON SI AVREBBE PIÙ un *random access iterator*, ma si avrebbe comunque un eccellente *bidirectional iterator* estremamente più ricco rispetto a quanto lo *standard* esige per quest'ultima categoria.   
  
Dato tutto ciò ormai per scontato, resta solo da dire che a un *output iterator* viene richiesto SOLAMENTE quanto si richiede a un *input iterator*, ossia alla categoria più debole di tutte, con l'UNICA aggiunta di avere ANCHE almeno UNA definizione di *operator=* membro, con un operando destro compatibile col tipo di *iterator\_traits<>::reference*, in modo da dare significato, e il significato atteso, a un'espressione come  
  
*\*iter++ = oggetto;*   
  
in cui ovviamente *iter* è un'istanza dell'*output iterator* e *oggetto* è quello che dice il nome stesso, ossia un oggetto di tipo *reference\_type*.  
  
Va ancora sottolineato che nell'espressione precedente non v'è in alcun modo la richiesta che dopo l'incremento e l'assegnamento l'iteratore *iter* sia ancora valido, a conferma di quanto detto nel paragrafo precedente. Se poi dovesse accadere ANCHE questo, significherebbe semplicemente che a diventare *mutable* è stato ALMENO un *forward iterator* anziché un semplice *input iterator*.  
  
Il canto si chiude segnalando l'esistenza di iteratori per le classi di *iostream*, ossia di iteratori capaci di svolgere il proprio lavoro avendo come contenitore sottostante uno *stream* sia di *input* sia di *output*; ne segue che l'operazione di incremento compiuta su tali iteratori si ripercuote sullo *stream* sottostante, cagionando, secondo la natura dello *stream* stesso, o l'estrazione o l'inserimento di un dato.  
  
Uno potrebbe domandarsi la ragione che ha indotto gli sviluppatori dello *standard* a profondere un impegno così vasto nella definizione di iteratori, giungendo addirittura a proporne di tali quali si è appena scritto. La risposta più immediata che mi viene in mente è: per uniformare le liste di chiamata di una dovizia di funzioni, definite negli *headers* chiamati *utility* oppure *algorithm*, che svolgono i più disparati compiti sui più diversi contenitori di tipi di dati, ricevendo per argomenti SOLO i corrispondenti iteratori in opportune istanziazioni di tali funzioni, ovviamente templatizzate.  
  
Trattandosi di iteratori come tutti gli altri, anche quelli per le classi di *ios* sono classi templatizzate, i cui nomi sono rispettivamente, indovinate un po': *istream\_iterator* e *ostream\_iterator*. Qui appresso se ne dànno le definizioni, così come sono reperibili nella sezione di *namespace std* inserita nell'*header* solito, ossia *iterator*:  
  
*template <  
class T,  
class C = char,  
class Cr = std::char\_traits<C>,  
class D = std::ptrdiff\_t  
> // fine della clausola di templatizzazione  
class // class, NON struct  
istream\_iterator // nome della classe, come ogni iteratore erede di...  
: public std::iterator<std::input\_iterator\_tag, T, D, const T\*, const T&>  
  
  
  
  
template <  
class T,  
class C = char,  
class Cr = std::char\_traits<C>  
> // fine della clausola di templatizzazione  
class ostream\_iterator   
: public std::iterator<std::output\_iterator\_tag, void, void, void, void>*   
  
Sono subito da notare le due diverse istanziazioni della classe madre *iterator* che fanno di *istream\_iterator* un *input\_iterator* e di *ostream\_iterator* un *output\_iterator* il quale, vivendo per giunta di vita propria e non avendo ALCUNA PRETESA di contribuire a rendere *mutable* qualsiasi altro iteratore, ADDIRITTURA istanzia come *void* TUTTI I TIPI contemplati per l'istanza *iterator\_traits<ostream\_iterator>* della *template iterator\_traits*. Inoltre anche la clausola di templatizzazione di *ostream\_iterator* è più corta rispetto a quella di *istream\_iterator*, mancando totalmente del quarto e ultimo parametro di templatizzazione di questa.  
  
Nel redigendo e futuro *standard* 2017 è già specificato che queste due classi non avranno più un'antenata: non so come questo fatto si ripercoterà sul buon programmatore.   
  
Se non state capendo una mazza, significa che **NON AVETE LETTO BENE QUESTO STESSO CANTO**: tornate quindi daccapo e rileggete dall'inizio.   
  
Chi è già tornato qui, oppure non se ne è mai andato, sappia che entrambe le classi sono dotate di costruttori, come è ovvio che sia, ed entrambe ne hanno precisamente DUE;  
  
*istream\_iterator* dispone di:

* un costruttore *default*, qualificato *constexpr* se l'istanza di *T* nella clausola di templatizzazione lo consente, che costruisce un oggetto *istream\_iterator* di natura equivalente a quello restituito dai metodi *end(  )* membri delle classi della *Standard Template Library*.
* un costruttore parametrico che riceve come unico parametro, e per riferimento, un oggetto di classe *istream*.
* oltre a questi due costruttori esplicitamente definiti, la classe marca con la parola di vocabolario *default* il proprio costruttore di copia.

*ostream\_iterator* dispone di:

* un costruttore parametrico che riceve come unico parametro, e per riferimento, un oggetto di classe *ostream*.
* un costruttore parametrico che riceve due parametri: il primo identico a quello ricevuto dal costruttore precedente, e il secondo che è di tipo *const C\**.
* essendo mancante tanto la definizione esplicita quanto la marcatura *default* del costruttore di *default*, la classe *ostream\_iterator* NON è *default constructible*, e ciò conferma il fatto che non può contribuire a rendere *mutable* alcun iteratore di categoria superiore a un *input iterator*.

Nel programma seguente, come sempre autocommentato, si fornisce un esempio d'uso di queste due classi:  
  
*# include <iostream>  
# include <sstream>  
# include <string>  
# include <iterator>  
using namespace std;  
  
int main(  )  
{  
// istanziazione di un input stream della classe istringstream  
// con contenuto misto: un double, una stringa e un int  
istringstream issa("1.4 tanto\_va\_la\_gatta\_al\_lardo 111 ");  
string str, stro;  
double d, do1;  
int i;  
  
  
// lettura ordinaria dello stream, come si è sempre fatto,  
// con ricopiatura ordinaria dei dati letti su output standard  
issa >> d >> str >> i;  
cout << d <<endl << str << endl << i << endl;  
issa . seekg(0);  
cout << "=============\n";  
  
  
// istanziazione di due istream\_iterator sullo stream issa  
// (riposizionato all'inizio, come opportuno)  
istream\_iterator<double> iter1(issa);  
istream\_iterator<string> iter2(issa);  
  
  
// lettura tramite istream\_iterator con assegnamento a variabili  
// di tipo congruo: la prima dereferenza serve per inizializzare  
// correttamente gli iteratori; poi si va per incrementi...  
do1 = \*iter1;  
stro = \*iter2;  
cout << do1 << endl;  
cout << stro << endl;  
do1 = \*++iter1;  
cout << do1 << endl;  
issa . seekg(0); // ancora riposizionamento all'inizio  
cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  
  
  
// istanziazione di due ostream\_iterator, uno su cout e uno su cerr  
ostream\_iterator<double> oter(cout, " questo viene scritto a ogni output\n");  
ostream\_iterator<string> oter\_(cerr, "\n");  
  
  
// sia la lettura sia la scrittura avvengono solamente tramite  
// l'incremento degli iteratori: appare chiara la funzione del  
// secondo argomento trasmesso ai costruttori di ostream\_iterator   
\*oter++ = \*++iter1;  
\*oter\_++ = \*++iter2;  
\*oter++ = \*++iter1;  
}*

**Canto trentanovesimo: i *threads***

*Così di ponte in ponte, altro parlando  
che la mia comedìa cantar non cura,  
venimmo; e tenavamo 'l colmo, quando  
  
restammo per veder l'altra fessura  
di Malebolge e li altri pianti vani;  
e vidila mirabilmente oscura.  
  
Quale ne l'arzanà de' Viniziani  
bolle l'inverno la tenace pece  
a rimpalmare i legni lor non sani,  
  
ché navicar non ponno - in quella vece  
chi fa suo legno novo e chi ristoppa  
le coste a quel che più vïaggi fece;  
  
chi ribatte da proda e chi da poppa;  
altri fa remi e altri volge sarte;  
chi terzeruolo e artimon rintoppa - :  
  
tal, non per foco ma per divin'arte,  
bollia là giuso una pegola spessa,  
che 'nviscava la ripa d'ogne parte.*E quale esordio più alto poteva avere questo canto, con cui il percorso di apprendimento del C++ si conclude? Anche la nostra *comedìa* non ha curato di cantare molte cose prima di giungere a questo passo, in cui ci apparirà *mirabilmente oscura* la misteriosa congerie dei cosiddetti *threads di esecuzione* o, semplicemente, *threads*.   
Anche l'immagine della frenetica attività invernale degli arsenali della Serenissima Repubblica di San Marco calza a pennello e perfino la *tenace pece*, al cui solo suono par di averla appiccicata addosso, e la *pegola spessa* fanno la loro figura e si adattano a quello di cui ci si accinge a discutere.  
  
In effetti, con l'introduzione nel linguaggio dei *threads*, avvenuta nel 2011, si apre a un programma la possibilità di eseguire alcune sue parti CONTEMPORANEAMENTE, sfruttando a fondo le architetture multi-processore dei calcolatori nostri contemporanei (bisticcio VOLUTO).   
Pertanto, se tornate con la mente al canto in cui si parlava di sequenzialità e *pietre miliari*, ORA capite che quei ragionamenti andrebbero completati aggiungendovi la frase *"tutto ciò a meno che non si introducano dei threads, perché, in tal caso, quanto è stato scritto si restringe al singolo thread ma non concerne un thread nei confronti di un altro"*.  
  
Quando, in un programma, si creano dei *threads* è come se l'esecuzione del programma stesso, che fino a quel momento procedeva spedita lungo un unico binario, incontrasse un ventaglio di scambi e quindi il binario si ramificasse in numerosi binari paralleli, tanti quanti sono i *threads*, tutti occupabili da singole parti diverse del codice. Allora veramente *chi ribatte da proda e chi da poppa*; veramente *altri fa remi e altri volge sarte*; veramente la carrozza di coda del convoglio del nostro programma può affiancare quella di testa e andare avanti ciascuna per suo conto per un periodo di tempo stabilito dal programmatore, ossia fino a quando tutti i binari originati al ventaglio di scambi si riuniscono novamente in un solo binario, con la ricomposizione corretta dell'intero convoglio.  
  
Se aveste prestato la dovuta attenzione, dovreste aver notato che è stato usato il verbo creare a proposito dei *threads*: in effetti essi non sono altro che OGGETTI (**Μεγαλη Γλωσσα** [scusate, ma stavolta andava detto in greco...]) e come tali possono essere creati in ogni momento dal buon programmatore come istanze di una certa classe, per l'appunto la classe *std :: thread*, definita, come si vede, nel *namespace* **std** e fruibile con l'inclusione *# include <thread>*.  
  
Per rendersi conto della portata di quello di cui si sta tentando di parlare è sufficiente considerare il seguente miserrimo programma, che costituisce per voi una sorta di viaggio nella memoria del tempo in cui eravate ancora piccoli (intendo piccoli rispetto all'apprendimento del linguaggio):  
  
*#include <iostream>  
#include <thread>  
  
namespace thread\_space  
{  
int s;  
}  
  
using namespace std;  
using namespace thread\_space;  
  
void f1(int n)  
{  
s = 11;  
for (int i = 0; i < 5; ++i) {  
cout  
<< "Io sono la funzione f1 con s = "  
<< s << " e n = " << n   
<< " \n" << flush;  
}  
}  
  
void f2(int& n)  
{  
s = 22;  
for (int i = 0; i < 5; ++i) {  
cout  
<< "Io sono la funzione f2 con s = "  
<< s << " e n = " << n  
<< " \n" <<flush;  
++n;  
}  
}  
  
int main(  )  
{  
int n = 0;  
f1(n+1);  
f2(n);  
cout  
<< "Alla fine n vale " << n  
<< " e s vale " << s << '\n';  
}*  
  
L'esecuzione del programma (l'avete eseguito?) non presenta la minima sorpresa: *f1* è invocata per prima e si esegue senza che di *f2* si abbia alcun sentore; quando questa sopraggiunge *f1* ha finito da un pezzo, e di lei non si sente più parlare. Gli effetti sulle variabili *n*, dichiarata in *main*, e *s*, dichiarata in un *namespace* del tutto equivalente a **std**, sono assolutamente scontati; l'inclusione *# include <thread>* non serve a niente.  
  
Ora osservate (**ED ESEGUITE!**) la seguente variante, compilandola con l'aggiunta dell'opzione *-pthread* al comando di compilazione che usate abitualmente:  
  
*#include <iostream>  
#include <thread>  
  
namespace thread\_space  
{  
thread\_local int s;  
}  
using namespace std;  
using namespace thread\_space;  
void f1(int n)  
{  
s = 11;  
for (int i = 0; i < 5; ++i) {  
cout  
<< "Io sono la funzione f1 con s = "  
<< s << " e n = " << n   
<< "\n" << flush;  
this\_thread::sleep\_for(chrono::milliseconds(0));  
}  
}  
void f2(int& n)  
{  
s = 22;  
for (int i = 0; i < 5; ++i) {  
cout  
<< "Io sono la funzione f2 con s = "  
<< s << " e n = " << n  
<< "\n" <<flush;  
++n;  
this\_thread::sleep\_for(chrono::milliseconds(0));  
}  
}   
int main(  )  
{  
int n = 0;  
thread t1(f1, n + 1);  
thread t2(f2, ref(n));  
t1.join(  );  
t2.join(  );  
cout  
<< "Alla fine n vale " << n  
<< " e s vale " << s << '\n';  
}*  
  
Alle due funzioni è stata solo aggiunta l'invocazione di una funzione che produce una sospensione fasulla dell'esecuzione della durata di 0 (zero!) millisecondi; serve a ingannare il compilatore, e indurlo a lasciarle eseguire ciascuna nel SUO *thread*, in modo da poter apprezzarne l'effetto: senza quella chiamata non si sarebbe visto proprio niente perché le due funzioni sono talmente semplici che il compilatore avrebbe pensato qualcosa come *"non scherziamo; qui mi si prende in giro"*... e così l'abbiamo preso in giro sul serio...  
  
Venendo alle parti significative del programma, le uniche variazioni SERIE si hanno in *main* e nel *namespace thread\_space* in cui la variabile *s* viene qualificata con la parola di vocabolario *thread\_local*.  
In *main*, al posto della semplice invocazione sequenziale delle due funzioni *f1* e *f2*, c'è appunto la creazione di due oggetti *thread*, ai cui costruttori sono trasmessi, nell'ordine, i nomi delle funzioni e i loro argomenti: si apprezzi l'uso della funzione *ref* per trasmettere al costruttore di *t2* il secondo argomento in modo che, grazie al *perfect forwarding*, *f2* possa riceverlo come lo richiede.  
  
Dovreste capire immediatamente che il costruttore parametrico di *thread* non può che essere una funzione templatizzata variadica (e come farebbe, se no?) e che il suo PRIMO argomento deve essere una classe templatizzata capace di essere istanziata sia con funzioni sia con classi a loro volta templatizzate (è o no un gran linguaggio?).  
  
Anche nella nostra semplicissima realizzazione si vede distintamente che le due funzioni sono eseguite simultaneamente e che alla fine, stavolta, *main* si trova la variabile *s* intonsa, dato che l'effetto della qualifica *thread\_local* consiste nel produrre una variabile distinta, chiamata sempre *s*, in OGNI *thread* che ne faccia uso: e *main* NON HA MAI reinizializzato la *s* del SUO *thread*.  
  
Le esecuzioni dei metodi membri *join* hanno lo scopo di reimmettere il binario parallelo sul binario principale: sono la ricomposizione del convoglio; solo quando OGNI *thread* ha eseguito il suo *join* il programma tornerà a proseguire normalmente e anche si dirigerà normalmente verso la sua naturale chiusura.  
  
Ora immaginate per UN SOLO ISTANTE che di *threads* diversi ce ne sia un intero contenitore, di quelli del canto scorso, pieno da traboccare, e che ciascuno svolga operazioni non così semplici come quelle delle nostre funzioni e magari anche in grado di influenzare quelle degli altri *threads*; immaginate anche di essere, come SIETE, il capostazione che dirige il traffico su questa complicata rete ferroviaria costituita dal vostro programma: di che cosa avreste bisogno?  
  
La risposta non è difficile: di posti di blocco, semafori e ferree regole di sincronizzazione e/o precedenza.  
  
Fortunatamente tutto ciò è disponibile (c'era dubbio?) nella classe *thread* attraverso i suoi metodi membri e le regole generali del linguaggio.  
  
Nel seguente esempio si fa cenno fugace a tutto ciò, sempre a un livello superficiale, ma non del tutto basso; e si mostra anche uno schema di interfacciamento dei *threads* con istanze di classi e con contenitori della *standard template library* come *vector*. Il codice, di per sé, non fa nulla di particolarmente interessante, ma mostra alcune soluzioni efficaci e anche l'uso di un *mutex*, ossia di un semaforo rosso che fa aspettare TUTTI gli altri *threads* fino a quando quello che l'ha acceso non lo spegne.  
  
*#include <iostream>  
#include <sstream>  
#include <vector>  
#include <iterator>  
#include <thread>  
#include <mutex>  
  
using namespace std;  
  
struct Ciccio  
{  
// qualsiasi congerie di metodi e variabili  
static unsigned int indice\_primario;  
unsigned int indice\_corrente;  
Ciccio(  ) {indice\_corrente = ++indice\_primario - 1;}  
};  
  
unsigned int Ciccio::indice\_primario = 0;  
  
mutex mutex\_;  
  
template <class X> struct Catena : vector<X>  
{static thread\_local Catena\* catena\_stazionaria;  
static Catena\* catena\_stazionaria\_di\_prima;  
using vector<X>::size;  
using vector<X>::begin;  
using ITERATOR = typename vector<X>::iterator;  
ITERATOR it;  
void algo(Catena\*& c, unsigned int i)  
{  
ostringstream os;  
catena\_stazionaria = new Catena;  
\*catena\_stazionaria = \*c;  
mutex\_.lock(  );  
os << "Sto eseguendo il thread " << i  
<< " che usa l'elemento # "  
<< (\*(catena\_stazionaria->it+i)).indice\_corrente  
<< endl << flush;  
mutex\_.unlock(  );  
clog << os.str(  ).data(  );  
delete catena\_stazionaria;  
}  
Catena(  )  
{  
it = begin(  );  
}  
void esegui(  )  
{catena\_stazionaria\_di\_prima = new Catena;  
\*catena\_stazionaria\_di\_prima = \*this;  
size\_t s = size(  );  
void (Catena::\*f)(Catena\*&, unsigned int) = &Catena::algo;  
thread \* threads = new thread[s];  
clog << "avviamento dei threads" << endl << flush;  
for(int i=0; i != s; ++i)  
threads[i] = thread(f, this, ref(catena\_stazionaria\_di\_prima), i);  
for(int i=0; i != s; ++i) threads[i] . join(  );  
clog << "threads eseguiti e riallineati con successo" << endl;  
delete catena\_stazionaria\_di\_prima;  
}  
};  
  
template <typename X> thread\_local Catena<X>\*   
Catena<X>::catena\_stazionaria;  
template <typename X> Catena<X>\* Catena<X>::catena\_stazionaria\_di\_prima;  
  
int main(  )  
{  
Ciccio ciccio[200];  
clog   
<< "creati " << Ciccio::indice\_primario  
<< " oggetti Ciccio" << endl;  
Catena<Ciccio> catena;  
for(auto i : ciccio) catena.push\_back(move(i));  
clog  
<< "catena detiene " << catena.size(  )  
<< " oggetti Ciccio" << endl;  
catena.it = catena.begin(  );  
catena . esegui(  );  
}*   
  
Dal programma appena scritto si impara quanto segue:

1. a ereditare un proprio contenitore da uno di quelli della *standard template library*; è il caso della *template struct Catena*, erede di *vector*, che consente la creazione di una *Catena<Ciccio>*.
2. il modo di riempire il contenitore, utilizzando il metodo *push\_back* ereditato, cui si trasmettono riferimenti destri agli elementi dell'*array* di *Ciccio*: ciò evita la generazione di copie di tali oggetti dato che ne viene eseguito il *move constructor*; d'altronde l'*array* dichiarato *ciccio* non serve a *main* se non come contenuto di *Catena*, che si incarica di gestirlo a dovere.
3. l'aggiornamento degli iteratori di *Catena* tramite la linea di codice  
   *catena.it = catena.begin(  );*  
   con cui ci si cautela da eventuali perdite di validità degli iteratori occorse in fase di riempimento del contenitore.
4. l'aver dovuto portare in ambito, durante la definizione di *Catena*, i metodi ereditati esplicitamente citati (*begin* e *size*) tramite le direttive *using*: *Catena* è una *template* e senza tali direttive il *name lookup* di *begin* e *size* sarebbe fallito (ANDATEVI A RILEGGERE il documento sul *name lookup*).
5. l'ulteriore apparizione della parola *using* come facente funzioni di *typedef* nella dichiarazione del tipo *Catena::ITERATOR*.
6. il fatto che la *struct Ciccio* mantenga traccia del numero delle proprie istanze in una variabile membro *static* (*indice\_primario*) e che OGNI istanza abbia una variabile membro (*indice\_corrente*) il cui valore ne rappresenta la collocazione ordinale nel contenitore *Catena*.
7. il fatto che il contenitore *Catena* abbia due membri *static* che sono puntatori alla stessa *template struct* (*catena\_stazionaria* e *catena\_stazionaria\_di\_prima*), il primo dei quali ha l'ulteriore qualifica *thread\_local*.
8. l'attribuzione del valore di *\*this* al puntatore *catena\_stazionaria\_di\_prima* da parte del metodo membro *esegui*; si tratta sostanzialmente di una fotografia istantanea scattata al contenitore nel momento in cui il metodo inizia a eseguirsi.
9. la dichiarazione, da parte di *esegui*, del puntatore a membro *f*, inizializzato coerentemente, rispetto alla segnatura del puntatore, con l'*offset* del metodo membro *algo* (accorciativo di *algoritmo*). Ed è in sostanza proprio la volontà o la necessità di aggiungere questo NOSTRO algoritmo che ci ha suggerito di rendere la *struct Catena* erede di *vector*.
10. l'allocazione, da parte di *esegui*, di un puntatore alla classe *thread* capiente di un numero di oggetti pari al numero di *Ciccio* contenuti in *\*this*; dato che questi oggetti sono stati costruiti dal costruttore di *default* della classe *thread* NON SERVONO **(ANCORA)** A NIENTE, ma non di meno **ESISTONO**.
11. l'assegnamento, a CIASCUN oggetto puntato dal puntatore *threads*, del valore di un oggetto INNOMINATO della sua stessa classe costruito dal costruttore parametrico con l'indicata segnatura: **È QUESTO IL MOMENTO** in cui i diversi *thread* si avviano SIMULTANEAMENTE, ciascuno sul proprio binario.  
    Deve essere ASSOLUTAMENTE tenuto presente che la classe *thread* **NON È COPIABILE** ed è per questa ragione che ne è stato richiesto il *move assignment operator* (ricordate? **RICORDATE!**). Tanto per essere ancora più chiari, una linea che recitasse  
    *threads[1] = threads[0];*  
    sarebbe **bollata dal compilatore, non come errore, ma come atrocità** (per inciso: CAPITE ADESSO il perché della *move semantic*?).
12. la segnatura del costruttore variadico di *thread* utilizzato, da confrontarsi con quella del precedente esempio: colà il primo parametro trasmesso era sempre stato il nome di una funzione definita nell'ambito globale; qui invece si tratta di un puntatore a un metodo membro di una classe. Tanto basta al costruttore, che sa riconoscere la differenza, per capire il significato del **secondo argomento** ricevuto, che dovrebbe essere ancora più chiaro a chi legge, e per concludere che nel thread che è chiamato a costruire deve essere eseguita la funzione  
      
    *this   ->\*  f(catena\_stazionaria\_di\_prima, i);*   
    ossia  
    *this   ->  algo(catena\_stazionaria\_di\_prima, i);*   
      
    proprio come se tale linea fosse stata esplicitamente scritta dal programmatore, a parte l'invocazione della funzione *ref*, necessaria per il **perfect forwarding** (ricordate? **RICORDATE!**), dato che il metodo puntato da *f*, ossia *Catena::algo*, riceve il suo primo argomento per riferimento.
13. il fatto che, subito dopo l'avvio dei *threads*, essi siano tutti ricomposti tramite l'invocazione del metodo *join* appartenente a ciascuno.
14. il fatto che, NEL FRATTEMPO, sono state eseguite, in maniera del tutto ASINCRONA, *size(  )* diverse versioni del metodo *algo*, TUTTE COMUNQUE APPARTENENTI ALL'OGGETTO *\*this*, il che significa che tutte avevano a propria disposizione lo STESSO INTERO CORREDO di variabili e metodi dell'oggetto...
15. ...eccettuati i membri qualificati *thread\_local*: di questi OGNI *thread* ha la propria PERSONALE ISTANZA, del tutto diversa e distinta dall'omologa istanza di quel membro detenuta da OGNI ALTRO *thread*. Significa che ciascuno di essi gestisce una versione distinta e separata del puntatore *catena\_stazionaria*. Queste distinte versioni si estinguono automaticamente con l'estinzione del *thread* di pertinenza.
16. il modo di lavorare tenuto dal metodo *algo*; non che sia l'unico possibile: ricordate l'assioma secondo cui il C++ non fornisce mai una soluzione UNICA.

Ci si soffermi appunto su *Catena::algo*, tenendo ben presente che ne esistono numerose versioni in esecuzione contemporanea, ognuna con un diverso valore ricevuto per l'argomento *unsigned int i*, rispecchiante quello della variabile *indice\_corrente* dell'elemento *i*-esimo di *\*this*.  
In questo modo *algo*, pur avendo a disposizione TUTTO il contenuto di *\*this*, è informata su quale degli elementi che *\*this* contiene debba rivestire per sé un significato, diciamo così, speciale.  
  
Non solo: *algo* prende anch'essa una fotografia istantanea di *\*this* prima di cominciare a lavorare, e la memorizza nella PROPRIA COPIA del puntatore membro *thread\_local catena\_stazionaria*. In questo modo non solo potrà essere in grado di confrontare il valore corrente di *\*this*, che magari lei stessa modifica, col suo valore iniziale, ma anche con eventuali modifiche apportate da (lei stessa in) altri *threads* (GRAN LINGUAGGIO).  
  
Tali modifiche potrebbero anche generare delle cosiddette *data races*, ossia situazioni conflittuali in cui (almeno) due *threads* pretenderebbero di accedere/modificare contemporaneamente lo stesso dato: il programma, nella versione presentata, non fa niente di tutto questo semplicemente perché non fa niente ai dati, ma, nella vita reale, perché non potrebbe accadere?.  
  
Ecco perché viene in soccorso l'oggetto *mutex\_* della classe *mutex*, che è una crasi di **mut**ual **ex**clusion: in questo programma è stato dichiarato nell'ambito globale per amor di brevità, ma evidentemente nulla vieterebbe di farlo diventare membro di appropriate classi.  
  
Quando un certo *thread* ne esegue il metodo membro *lock* quel *thread* continua la propria esecuzione sospendendo quella di OGNI ALTRO *thread* concorrente. È il semaforo rosso per tutti gli altri binari. Il verde scatta SOLO quando lo stesso *thread* esegue il metodo *unlock*: è evidentemente indispensabile NON DIMENTICARSI DI ESEGUIRLO, altrimenti il programma si mette da solo in una situazione peggiore di quella di Falstaff e Ford quando dovevano uscire dalla porta dell'osteria della giarrettiera.  
  
Anche nella versione presentata c'è un minimo rischio di incorrere in quella situazione, ossia semaforo rosso su TUTTI I BINARI, qualora TUTTI i *threads* eseguissero *mutex\_lock(  );* nello STESSO, IDENTICO, *clock* di macchina: una situazione, invero, con probabilità ridicola di accadimento; niente paura; esistono anche dei *mutex* temporizzati che diventano verdi COMUNQUE allo scadere di un tempo prefissato dal programmatore; e in quel caso diventa essenziale poter fare dei confronti sui dati sensibili per evitare di riavviare il convoglio del programma non si sa come. Ed ecco a che cosa può servire aver preso delle fotografie...  
  
Dato che, tra l'esecuzione di *mutex\_.lock(  );* e quella di *mutex\_.unlock(  );*, il *thread* che le esegue assume l'esclusiva della gestione dell'intera memoria del programma, sarà compito del buon programmatore far eseguire lì quelle operazioni che avessero la potenzialità di generare *data races*, realizzando in tal modo il compromesso MIGLIORE che contempli SIA la loro esclusione SIA la velocizzazione spudorata dell'esecuzione del codice.  
  
Nel nostro esempio l'effetto apportato dall'utilizzo del *mutex* è immediatamente apprezzabile semplicemente togliendolo: rieseguite dunque il programma dopo aver commentato le richieste di esecuzione SIA di *mutex\_.lock(  );* SIA di *mutex\_.unlock(  );* e aver sostituito *os <<* (s'intende in *algo*) con *clog <<*, e poi venite a riferirmi, se volete, che cosa vi è successo.  
  
Un'ultima annotazione sul modo di gestire l'*output* da parte di *algo*; premesso che, in generale, una funzione *threadizzata*, consentitemi il neologismo,  
  
**MENO *OUTPUT* FA MEGLIO È, e al limite farebbe molto bene a starsene ZITTA**  
  
(prendetelo come un assioma) e assodato che COMUNQUE è sempre meglio fare *output* provvisori in memoria, riservandosi l'uscita sui canali *standard* solo quando l'*output* stesso è terminato, dovrebbe risultare chiaro, alla luce anche di quanto avverrebbe usando *clog* senza *mutex* (**l'avete fatto l'esperimento?**), il motivo per cui si è scelto di adottare la soluzione che comporta l'uso di un *ostringstream*, visto che la funzione, dovendo servire da esempio, bisogna pure che faccia qualcosa. In effetti, mentre *clog* è visibile in ugual misura a TUTTI i *threads*, l'oggetto *os* della classe *ostringstream* pertiene in esclusiva ad *algo*, essendo dichiarato nel SUO ambito e quindi è DIVERSO tra un *thread* e l'altro.   
  
Ma è tempo ormai di ormeggiare la *piccioletta barca* e bersi un ottimo punch al rhum.  
  
Se voleste salpare di nuovo per andare a esplorare il continente dei *threads* fin nella giungla più fitta e fino a civilizzare i cannibali che vi potreste incontrare ... e magari andare a scoprire le terre nuove che si trovano al di là degli oceani degli editandi *standard* 2014 e 2017 ... non avete che da richiamare il nostromo...finché dura...  
  
*A l'alta fantasia qui mancò possa;  
ma già volgeva il mio disio e 'l velle,  
sì come rota ch'igualmente è mossa,  
  
l'Amor che move il sole e l'altre stelle.*