Note | C'hi++

Questo libro è dedicato ai miei nipoti.
I tre che vivono in Francia e i tre miliardi che vivono anche altrove.

Indice

I linguaggi di programmazione

Cosa disse Brian Kernigan - Un computer è come una nave - Come tornare al tuo albergo in Cina - I paradossi sono la crittografia di Dio - PacMan non paga le tasse.

Il C++

Essere come il C++ - L'astrazione dei dati, la programmazione a oggetti e la monta equina - Un buon programmatore non si accontenta - La differenza fra C e C++, secondo Stroustrsup.

I commenti

I commenti come la letteratura - Un anticipo sui Post-It.

I tipi di dato

L'Universo è fatto di spazioni - Il Mondo ha la natura dell'Arte - René Guenon aveva ragione - Perché la luce non può andare più veloce.

Struttura dei programmi C++

La funzione main del C'hi++ - Le forze che muovono l'Universo: Gravità, Entropia e Annosa Dicotomia - Chi creato l'Universo - Cosa dice quel senza-Dio di Dawkins.

Gli operatori

I concetti di vero e di falso sono frutto di un arbitrio - William James ha ragione - Fama, successo e prestigio - Le certification authority etiche - Affinità fra le persone per bene e i malandrini.

Il preprocessore

Valori entropici e valori gravitazionali - La pornografia per il Giudice Potter Steward - L'amore è una backdoor - Il Cielo non è interessato alla tua felicità - L'obbligo di ringraziare.

La memoria

La gestione della memoria - Il rapporto degli esseri umani con l'operatore delete - Le Cinque Fasi di elaborazione del lutto della Società moderna - Un precetto sull'amore.

Le funzioni

Un suggerimento di Jacopone da Todi - Il problema dei romanzi - Le regole di vita di Myamoto Musashi - La Verità è una modella - La Via del carpentiere.

Istruzioni condizionali

Come decidere cosa sia giusto e cosa no - I roghi, prima e dopo il 6 Giugno del 1945 - L'importanza del pentimento - Perché Nansen potè tagliare in due il gatto

Istruzioni iterative

Scòpo delle religioni - I due dogmi del C'hi++ - Il versionamento del software e delle filosofie - Gli scienziati non sono infallibili.

Classi e oggetti

Affinità fra Platone e il C++ - Perché ciò che amiamo ci ucciderà (se tutto va bene) - Libero arbitrio e dissonanze nel Jazz - Cartesio non capiva nulla di serigrafia

L'ereditarietà

Il Karma delle funzioni virtuali - La Scienza è transeunte; il Mito è immortale - Perché 13 e 17 portano sfortuna - Cos'è l'Arte - Le regole dell'Ikebarba

Il polimorfismo

Una metafisica non metafisica - Bug noti delle religioni e degli esseri umani - Affinità fra il C'hi++ e le filosofie canoniche - Mappe topografiche e immagini da satellite - La Banda degli Onesti.

Gli stream

Problemi esistenziali del codice di esempio - Un hard-disk meta-fisico - Mezzo secolo di fallimenti ideologici - Etica degli Scacchi - John Lennon se l'è cercata.

Il debug

Gli errori sono inevitabili - Come correggere i bug della nostra vita - Chiedere scusa e chiedere perdòno - I fattori che influenzano il nostro comportamento - Non si devono adorare le parole

Epilogo

La morte del Maestro, nella finzione e nella realtà

Ringraziamenti

Le persone senza le quali C'hi++ non sarebbe stato scritto. (Decidete voi se sarebbe stato un bene o un male.)

Bi(bli)ografia

I libri e gli editori senza i quali C'hi++ non sarebbe stato scritto. (v. sopra)

Note

Note al testo e una nota (con note) sulle note.

Prologo

In principio era il login

Nacqui da famiglia ricca, ma troppo tardi.
Secondogenito, vidi la florida impresa paterna andare in dote — per diritto di nascita, ma anche per naturale inclinazione — ai miei monozigotici fratelli maggiori e, com'è consuetudine per i figli cadetti, fui avviato alla vita monastica. Entrai in seminario all'età di nove anni e presi i voti il giorno del mio diciottesimo compleanno. Conobbi il Maestro quattro anni dopo.
Fui mandato da lui perché un vecchio programma, dopo quasi un decennio di attività ininterrotta, aveva avuto dei problemi e non c'era nessuno nel monastero che sapesse come porvi rimedio. Pur avendo trascorso in quello stesso edificio due terzi della mia vita, non avevo mai incontrato il Maestro, prima di allora. Sapevo poco di lui, principalmente che non era bene saperne molto e non l'avevo mai visto né nel refettorio, né in occasione di celebrazioni o emergenze. Era, per tutti noi giovani, una sorta di figura mitologica, maligna o benigna a seconda dei racconti che su di lei si ascoltavano.
Bussai alla sua porta con cautela, quasi mi aspettassi terribili rappresaglie per quell'indebita intromissione nella sua vita e con altrettanta cautela accettai il suo invito a entrare. L'arredamento delle nostre stanze era semplice ed essenziale, ma in confronto a quello della cella del Maestro, appariva sfarzoso. Non c'erano finestre alle pareti e le uniche due fonti di luce erano una piccola lampada e lo schermo di quello che sembrava un vecchio computer risalente alla fine del XX secolo. In un angolo c'era un curioso divano-letto, simile alle panchine dei parchi, che doveva essere il suo giaciglio per la notte; al lato opposto della stanza, una sorta di libreria realizzata con mattoni e assi di legno grezzo sorreggeva i suoi pochi effetti personali. Nulla copriva la pietra del pavimento.

«Il mio programma sta dando degli errori.»
Risposi di sì, anche se la sua era stata un'affermazione. Il Maestro annuì.
«Era stato istruito a farlo. Adesso lo mettiamo a posto.»
Cominciò a battere sulla tastiera del suo computer e io mi avvicinai incuriosito. Quando arrivai a vedere l'immagine proiettata dallo schermo capii perché nessuno all'interno del convento sapesse intervenire sul suo programma.
«Sì, è C++,» disse il Maestro, intuendo la mia curiosità. «Un tempo si pensava che fosse il linguaggio del futuro. Come di tutti i linguaggi, del resto.»

Chiesi cosa volesse dire che il programma era stato istruito a dare errori.
«Questa è una domanda sciocca. D'altro canto, il fatto che stessi dicendo la verità è evidente, visto che non ti ho nemmeno chiesto che genere di difetto fosse stato rilevato. Comunque, puoi andare a dire al tuo superiore che il vecchio pazzo ha corretto l'errore.»
Non riuscivo a distogliere lo sguardo dallo schermo, ero come ipnotizzato da quello strano codice che potevo capire solo in parte. In quelle sequenze di istruzioni c'era qualcosa che non avevo mai visto prima: un ritmo, una sorta di indefinibile bellezza di cui mi ero innamorato a prima vista.
Chiesi al Maestro di insegnarmi il C++.
«E perché mai? ci sono linguaggi molto più facili da usare.»
Gli spiegai che non si trattava di un interesse tecnologico, ma estetico. Lui restò in silenzio per qualche secondo, considerando quello che avevo detto, poi chiese:
«Qual'è il dovere di un programmatore?»
Lo pregai di definire meglio la sua domanda.
«Torna qui domani; se mi saprai dire qual'è il dovere di un programmatore, ti insegnerò il C++.»

Passai tutta la notte a meditare su quella strana domanda e la mattina dopo mi presentai al Maestro. Dissi che il dovere di un programmatore era quello di scrivere del buon codice. Il Maestro non distolse nemmeno lo sguardo dallo schermo e disse:
«Torna qui domani; se mi saprai dire cosa vuol dire scrivere del buon codice, io ti insegnerò il C++.»

Com'è facile intuire, anche il giorno dopo e per diversi giorni a seguire il Maestro trovò il modo di rimandare l'inizio del mio tirocinio con domande ancora più specifiche che andavano a colpire anche la più lieve lacunosità delle mie risposte. Analizzai ogni possibile aspetto della produzione del software, dall'utilizzo delle risorse di sistema alle implicazioni sociali dell'incremento dell'occupazione che deriva dall'evoluzione dei programmi, ma non ci fu nulla da fare: ogni volta il Maestro riuscì a trovare una scappatoia per venire meno al suo impegno.
Alla fine non ne potei più. Esasperato, dissi che ne avevo abbastanza di quella sua ostinata capziosità: per quanto io potessi essere specifico nelle mie risposte, ci sarebbe sempre stato un margine di indeterminazione. Se voleva insegnarmi il C++ doveva iniziare quel giorno stesso.
Il Maestro mi fissò e, sorridendo, spense il computer.
«No, per oggi basta. Vieni domani per la seconda lezione.»

Negli ultimi mesi la salute del Maestro è peggiorata e così gli ho chiesto il permesso di trascrivere i suoi insegnamenti per poterli trasmettere a mia volta ad altri discepoli.
Con mia grande sorpresa, ha accettato.

Incipit

> *.* è il migliore

Quello che stai facendo è inutile.
Voglio che tu lo scriva all'inizio del tuo libro, così la gente non perderà tempo a leggerlo. Tutto ciò che si fa con uno scòpo è inutile. In particolare i libri, perché sono stupidi: non si accorgono di quando sbagliano e anche se se ne accorgessero non ci potrebbero fare nulla: seguiterebbero a dire una cosa sbagliata. Tu pensi di scrivere una nuova Bibbia: anche fosse, sei sicuro di volerlo fare? Tutti i libri sacri partono da un presupposto sbagliato: che valga la pena scriverli, perché l'Umanità può essere salvata e, soprattutto, che merita di essere salvata. Non è così. Credi che se gli autori della Bibbia o dei Vangeli avessero potuto prevedere le conseguenze ultime delle loro opere — le guerre, le morti, le sofferenze —, li avrebbero scritti lo stesso? I libri sacri sono sempre stati strumentalizzati dai potenti, cosa ti fa pensare che per il tuo non succederà lo stesso?
Il Maestro Musashi, nel suo *Dokkodo*, scrisse:

> Fà che le future generazioni non siano legate ad armi antiche.

Tu proietti su questo linguaggio ormai morto delle aspettative personali, speri che possa far rivivere il tempo passato e ti illudi che il tempo passato sia meglio di quello presente. Ti sbagli. Il passato, mio giovane amico, è una vecchia amante che col suono della sua voce ci ricorda i piaceri vissuti insieme, ma che bada a restare alle nostre spalle per nasconderci il suo volto rugoso.

Io ti insegnerò il Chi++, non ti indicherò la Via. Quando qualcuno ti indica la Via, ricordati sempre che una via la si può percorrere in due direzioni: una direzione ci avvicina alla nostra méta; una direzione ce ne allontana. La direzione è importante, non la via. Se un uomo conosce la direzione, la via non ha più alcuna importanza.
Il C'hi++ è allo stesso tempo un linguaggio di programmazione, un atteggiamento mentale e un modo di vivere. Chi pratica il C'hi++ scrive codice come se stesse vivendo e vive come se stesse scrivendo del codice.
Vivere è come scendere in canoa lungo un fiume. Affannarsi a risalire la corrente è inutile e infruttuoso, perché il fiume è più forte di noi e non si stanca mai. Lasciarsi andare alla corrente è pericoloso, perché ci si potrebbe parare davanti un ostacolo e noi non avremmo modo di evitarlo. È necessario quindi remare solo quel tanto che ci permette di essere più veloci della corrente e di schivare gli ostacoli che, di volta in volta, si presentano sul nostro cammino.
Similmente, scrivere del buon codice significa scrivere solo il codice necessario a raggiungere lo scòpo che ci si è prefissi. Aggiungere una sola virgola in più è sbagliato, perché rende più difficile il debug e più lento il programma. Chi pratica il C'hi++ applica lo stesso principio alla sua vita e compie solo le azioni necessarie, ignorando tutto ciò che è superfluo. Capire quali sono le azioni necessarie è semplice, così com'è semplice capire qual è la direzione della corrente

quando navighi su un fiume. La cosa difficile è capire di essere su un fiume.
Tutte le cosmogonie che sono state formulate nel corso dei secoli hanno una caratteristica comune: richiedono una certa dose di fede in qualcosa di cui non è possibile provare l'esistenza; uno o più elementi fittizi e sovra-naturali, che si introducono nel sistema per spiegare ciò che non è possibile dimostrare praticamente. Credere in queste entità richiede l'accettazione di condizioni più paradossali di quelle che si cercava di spiegare originariamente. Ma, come diceva sempre il Maestro Canaro: "la Verità è semplice" e qualunque spiegazione che richieda dei parametri correttivi è, per definizione, inesatta.
Ciò che le religioni non dicono, comunque, è che non si può andare in Paradiso da soli: o ci andiamo tutti, o non ci va nessuno.

C'hi, in giapponese, vuol dire "respiro". L'Universo è scritto in C'hi++. L'Universo respira. Ciò che noi percepiamo come eternità altro non è che un singolo respiro dell'Universo, una delle innumerevoli eternità che si sono succedute dall'inizio dei tempi. Ogni respiro dell'Universo comporta una fase di espansione e una fase di contrazione. I punti terminali di ciascuna fase prevedono l'annichilimento di ogni forma di vita senziente. Al contrario, nei periodi intermedi nascono e muoiono innumerevoli mondi. Su ciascuno di questi innumerevoli mondi nascono e muoiono innumerevoli esseri Ciascuno di questi innumerevoli esseri, nel corso della sua vita, compie una serie di azioni che influenzano la sua esistenza e quella dei suoi simili. Le possibili permutazioni di queste innumerevoli azioni per questi innumerevoli esseri per questi innumerevoli mondi sono quasi illimitate. Quasi. Il tempo, al contrario, è illimitato, quindi può capitare che una determinata sequenza di eventi, già avvenuta in un precedente respiro dell'Universo, si ripeta, con condizioni simili, in un respiro successivo. Ciò vuol dire che, con buone probabilità, io e te abbiamo già avuto questa conversazione in una o in migliaia delle nostre esistenze precedenti.

Mitopoietica del C'hi++

Il settimo giorno, Dio fece il backup

Su di me ci sono due teorie: qualcuno pensa che io sia impazzito cercando di fare il *debug* dell'Universo, altri pensano invece che io abbia deciso di fare il *debug* dell'Universo perché sono pazzo.

Anche il Maestro Canaro fu accusato di essere pazzo quando disse di essere stato incaricato da Dio di fare il *porting* dell'Universo dal COBOL al C++, ma non era pazzo. Ho conservato questi documenti: sono l'ultima pagina del suo diario e le relazioni che furono inviate alla società per cui lui lavorava mentre

era distaccato dal “Cliente”. Mettili all’inizio del tuo libro, così tutto ciò che diremo dopo non sarà che una ripetizione.

Ultima pagina del diario del Maestro Canaro (8 Aprile)

Sono in ufficio e sto lavorando quando squilla il telefono. La voce che mi risponde è piuttosto bassa e ha un'eco curiosa. Mi fa:
- Canaro, sono Dio, ho bisogno di te.
- Marco, ho da fare, ci sentiamo dopo.
Riattacco. Un attimo dopo il telefono squilla di nuovo. Stavolta faccio caso al tipo di squillo e deduco che si tratta di una telefonata esterna.
- Canaro, non sono il tuo collega, sono davvero Dio. Devi aiutarmi a fare il porting dell'universo in C++.
Non posso fare a meno di sorridere.
- Carina. E anche il sintetizzatore vocale mi sembra che funzioni piuttosto bene. Senti, finisco questa relazione e ti offro un caffè, ma adesso lasciami lavorare.
Riattacco di nuovo e di nuovo il malefico oggetto fa sentire la sua voce. È un'esterna, non posso rischiare: potrebbe essere un cliente, ma so già che..
- Canaro, ascoltami una buona volta: sono davvero Dio e ho bisogno che tu mi risolva un problema.
Sto per dirgli qualcosa di cui poi mi pentirò, ma la voce continua:
- Se non mi credi, stacca la spina del telefono.
- Ma per favore..
- Staccala!
Più per curiosità che altro, vado a vedere il suo bluff. Stacco la spina.
- Adesso mi credi?
Devo dire che per un attimo il mio scetticismo vacilla. Cerco una possibile spiegazione.
- Hai messo una ricevente nella cornetta.
- Testardo eh? Va bene: scegli un oggetto a caso.
- Per fare cosa?
- Per utilizzarlo come cornetta, uno che il tuo collega non possa aver manomesso.
- Marco, per favore..
- Avanti, cosa ti costa? Visto che sei così sicuro..
- La mia pipa va bene?
- Quello che vuoi: avvicinala all'orecchio.
Vediamo dove vuole andare a parare: tiro fuori la mia pipa dall'astuccio di pelle e avvicino il fornello all'orecchio.
- Fatto, - dico e intanto mi guardo intorno in attesa dello scherzo.
- Lo so, - la voce è nel fornello di radica. - Adesso, se per favore potessimo riprendere la nostra conversazione telefonica.. Non vorrei

che entrasse qualcuno e ti vedesse parlare con una pipa.

- Premetto che io di queste cose di computer non ci capisco niente,
- mi dice appena ho riattaccato la spina del telefono.
- Mio figlio, invece, i computer li sa usare e mi ha detto che sarebbe meglio fare il porting dell'universo in un linguaggio a oggetti, suggerendomi questo C++. Ora, vedi, io vado forte in scienze naturali, non in informatica, e tutti questi termini tecnici, per me, sono misteriosi: cosa vuol dire fare il porting di qualcosa? e cosa sono i linguaggi orientati agli oggetti?

"Bella domanda", pensai. Anni prima avevo scritto un manuale di C++ e il paradigma a oggetti era l'argomento del primo capitolo. Se solo lo avesse letto..

- Ho anche dato una scorsa al tuo libro, ma non ci ho capito niente: scrivi di schifo, lasciatelo dire. E poi, mi sono detto, chissà quante cose sono cambiate nel frattempo, così ho deciso di chiamarti.
- Innanzitutto la ringrazio per i complimenti, poi, per venire alla sua domanda, fare il porting di qualcosa vuol dire riscrivere un'applicazione con un linguaggio di programmazione differente da quello in cui era stata scritta inizialmente.
- E questo a quale scopo?
- Di solito per sfruttare le potenzialità del nuovo linguaggio o per poter utilizzare l'applicazione in un sistema per cui il vecchio linguaggio non sia valido.
- Come tradurre i dieci comandamenti dall'Ebraico al Latino?
- Qualcosa di simile.
- E questo "C++"? Cosa vuol dire "linguaggio orientato agli oggetti"?
- Che invece di tipi di dato astratti come numeri e caratteri, permette di utilizzare dei tipi di dato simili a quelli reali.
- Non ho capito.
- I vecchi linguaggi di programmazione costringevano il programmatore a utilizzare solo dei tipi di dato predefiniti: numeri, lettere, date. Questo va bene per un certo tipo di applicazioni, ma rende difficile scrivere applicazioni più complesse dove può essere utile sfruttare un linguaggio che sia capace di gestire anche dei nuovi tipi di dato, definiti dall'utente. In C++, il linguaggio che suo figlio le ha consigliato, questo si ottiene per mezzo delle classi.
- Come in zoologia?
- Una specie. Per classe si intende un tipo di dato non standard, del quale si può definire il comportamento.
- Fammi un esempio, che non capisco. E poi basta con questo lei, passiamo al "tu", che è più semplice.
- Mettiamo che tu abbia due stringhe..
- Non porto scarpe con le stringhe, uso i sandali Birkenstock.
- Intendevo delle sequenze di caratteri. Ora, per come un computer

gestisce i dati normalmente, non ha senso addizionare due stringhe, dato che non si tratta di un tipo di dato numerico, ma si può definire una classe "Stringa" che, se sottoposta ad addizione, faccia qualcosa di sensato.
- Del tipo?
- Poniamo che tu abbia una stringa A qualsiasi.
- "Pippo" va bene?
- Originale. E che tu voglia unirla ad un'altra stringa B.
- "Pluto".
- Anche questa bella originale. Comunque, tu scriverai l'istruzione: C = A + B e nella stringa C ci sarà scritto: "PippoPluto".
- Lo terrò a mente la prima volta che mi capiterà di dover scrivere: "PippoPluto".
- Era un esempio.
- Seguito a non capire perché si chiamino "linguaggi orientati agli oggetti".
- Perché non gestiscono dati grezzi, ma oggetti con un loro comportamento ben definito.
- Devo vederlo in pratica. Comunque, quali sarebbero i vantaggi di questa traduzione?
- Nel caso del C++, una maggior facilità di analisi.
- Ma l'analisi dell'applicazione già c'è.
- E una maggiore facilità di debug.
- Prego?
- Di correzione degli errori.
- L'applicazione attuale è in funzione da un'eternità e non ha mai dato nessun problema.
- Aumenta anche le possibilità di riutilizzare il codice in altre applicazioni.
- Questo mi sembra utile quanto la stringa "PippoPluto".
- Il processo produttivo ha una certificazione di qualità?
- Io ho visto che era buono: direi che può bastare.
- Be', allora non vedo che necessità ci sia di riscriverla.
- Esattamente quello che ho detto a mio figlio, ma lui ha cominciato a parlare di questo C++ e di quell'altro, quello col nome di un'isola, come si chiama..
- Java? Non si riferisce a un'isola, ma a un tipo di miscela per il caffè.
- Quello che sia. Il discorso è: che bisogno c'è di riscrivere tutto?
- Per quello che ne posso capire, direi nessuno..
- Bene, ti ringrazio.
Attacca (o quello che è), ma dopo un po' richiama.
- Canaro, ho parlato con mio figlio e lui dice che sono un retrogrado, che l'applicazione attuale è obsoleta, che gli utenti si lamentano e che questo porting è necessario. Io non ci capisco più niente, cosa devo fare?

Sto per rispondere ”Lo sa Iddio”, ma fortunatamente mi trattengo e ricorro alla risposta standard in questi casi:
- Potremmo fare uno studio di fattibilità e poi decidere..
- Mi sembra una buona idea. Quando puoi cominciare?
- Non lo so, devo chiederlo al mio responsabile.
- D'accordo: chiediglielo e poi fammi sapere.
Riattacca senza dirmi come potrò ”farglielo sapere”, ma questo è l'ultimo dei miei problemi. Il primo, al momento, è in che modo raccontare questa cosa al mio capo senza farmi prendere per pazzo.

- Se speri di rimediare un periodo di convalescenza per esaurimento nervoso, scordatelo. Hai detto (e soprattutto fatto) cose molto più strane.
- Lo so che è pazzesco, ma è vero: Dio mi ha telefonato e mi ha chiesto di riscrivere l'Universo in C++.
- Canaro, piantala e vai a finire l'analisi, che la dobbiamo portare al cliente nel pomeriggio.
- E se mi richiama, cosa gli devo dire?
- Raccontagli di quanto hai sbagliato le stime dell'ultimo progetto, così si rivolgerà a un altro.
Scornato, me ne torno al mio loculo dove mi preparo in anticipo la seconda pipa della giornata. Mentre armeggio con il tabacco, il telefono squilla per l'ennesima volta, ma non è chi mi aspetto che sia, bensì il mio capo che mi richiama nel suo ufficio. ”Chiamata di re, tanto buona non è..”, penso, ma quando arrivo nella ben nota stanza dall'inequivocabile odore di sigaro toscano, mi trovo di fronte un uomo d'umore molto diverso da quello che mi ha allontanato poco prima. La voce è pacata, i modi gentili e mostra la massima comprensione e disponibilità nei miei confronti.
- Passa le cose che stai seguendo a Igino e Paolo e dedicati completamente allo studio di fattibilità per il porting dell'Universo. Stupito per il repentino voltafaccia, chiedo la prima cosa che mi passa per la testa.
- Su quale commessa devo scaricare le ore?
- Ancora non lo so, ma non te ne preoccupare. Oh, se ti serve un portatile puoi prendere il mio.
- Va bene, grazie. C'è nient'altro?
- Tieni, - mi porge un foglietto con qualcosa scritto sopra. - ha detto che puoi richiamarlo a questo numero.
Rientro nel mio ufficio, chiudo la porta e compongo il numero: una voce registrata mi avvisa che l'utente è al momento occupato, ma che è stato inviato l'avviso di chiamata. Non passa un attimo che sento la Sua voce dire:
- Sì, pronto?
- Scusami, io ti ho richiamato subito, ma se sei al telefono..
- Non ti preoccupare, stavamo chiudendo. Allora?

- Sono a tua disposizione, ma prima di iniziare c'è una cosa che devo dirti.
- Vuoi sapere come ho fatto a convincere il tuo capo? È stato facile, gli ho detto che a prendere le cose sul serio e ad arrabbiarsi ci si accorcia la vita.
- Questo credo che lo sapesse già..
- Probabile, ma io gli ho detto di quanto.
- Tu sai quando morirà?
- Teoricamente sì, ma chi se lo ricorda.. Ho sparato una data a caso entro la fine dell'anno. Gli ho detto anche che questa era la sua ultima possibilità per far sì che quel doloretto che ha sentito al torace stamattina appena sveglio fosse solo un doloretto. È bastato.
- Almeno così sembra. Comunque non è questo che volevo chiederti.
- Oh! Tu probabilmente vuoi sapere perché ho scelto proprio te, avendo a disposizione i migliori.
- Io avrei detto "fra i migliori", ma comunque..
- Te lo dirò se farai un buon lavoro. Noi ci vediamo domani mattina all'indirizzo che ti ha dato il tuo capo. Cerca di essere puntuale e non passare dal centro perché ci sarà una fila per un tamponamento.
- Me ne ricorderò.

Prima relazione sull'andamento del progetto "U++" (19 Aprile)

Le cose vanno meglio di quello che sperassi: malgrado i suoi impegni, il Cliente trova sempre del tempo da dedicarmi e il mio lavoro procede spedito. Inoltre, a dispetto di una malcelata ritrosia iniziale, credo che finalmente si sia convinto dell'utilità di una revisione del codice e adesso è Lui stesso, e non più il Figlio, a spingere per una rapida conclusione. Due sono state finora le scoperte di maggior rilievo: la prima è che l'applicazione attuale è scritta in COBOL (il Cliente mi ha spiegato che l'ipotesi iniziale di utilizzare il FORTRAN era stata abbandonata dopo un primo tentativo e che in quel linguaggio è stato scritto solo il cervello degli ingegneri); la seconda è che Mandelbrot aveva ragione: l'Universo ha una struttura frattale (scelta, questa, che semplifica di molto la gestione di situazioni complesse tipo l'interazione fra gli agenti atmosferici, il rumore dei pedalò e gli incontri casuali con ex amanti nei grandi magazzini). Una mia proposta che ha incontrato l'approvazione del Cliente è di procedere per gradi, stendendo un'analisi generale completa, ma riscrivendo l'applicazione poco per volta, in modo da produrre il minimo disturbo possibile agli utenti e semplificando il lavoro di affinamento delle procedure. Ho scritto una relazione preliminare (che allego) e l'ho consegnata al Cliente: domani mi farà avere una sua opinione al riguardo.

Seconda relazione sul progetto "U++" (20 Aprile)

Nella mattinata odierna ho discusso la mia relazione preliminare con il Cliente, che nel complesso mi è sembrato soddisfatto dell'impostazione prescelta. Dopo qualche tentennamento, si è anche convinto che non è possibile fare dei paragoni fra i tempi di sviluppo dell'applicazione precedente e di quella nuova, sia per l'impossibilità di paragonare analisi strutturata e analisi orientata agli oggetti, sia per la difficoltà di comparare la misurazione attuale, in giorni/uomo, con quella precedente, in giorni/Dio. Al momento ho solo una preoccupazione, ovvero che qualcuno, da dietro le quinte, piloti gli umori del Cliente con fini non chiari. Dico ciò, perché tutti i dubbi e le perplessità da lui espresse finora sono largamente al disopra delle sue conoscenze tecniche. Se inizialmente pensavo che dietro a tutto questo ci fosse quel Figlio di cui ho già fatto menzione nella memoria inviatavi prima della mia partenza (il che mi avrebbe posto in una situazione oltremodo delicata in quanto, come tutti ben sappiamo, la mia stessa presenza qui è dovuta proprio ad un suo suggerimento e metterne in discussione la competenza sarebbe controproducente), ora sono ragionevolmente certo che si tratti di qualcun altro, anche se non so proprio né chi possa essere né quali possano essere i suoi scopi.

Terza relazione sul progetto "U++" (28 Aprile)

Mi scuso per il prolungato silenzio, ma le riunioni degli ultimi giorni sono state letteralmente massacranti e non ho avuto il tempo di mantenervi aggiornati sui più recenti sviluppi della situazione. Di contro, va detto che finalmente comincio ad avere un'idea un po' più chiara delle diverse esigenze funzionali e dei ruoli coinvolti nel progetto, il che è sicuramente un bene, specie perché il terreno, qui, comincia a scottarmi sotto i piedi. E non faccio per dire. Come certo ricorderete, nella mia precedente comunicazione avevo paventato l'esistenza di ingerenze occulte e, comunque, un dubbio che avevo avuto sin dall'inizio era: dato che il Cliente ha più volte affermato di non possedere grosse cognizioni tecniche in campo informatico, chi ha scritto a suo tempo la prima applicazione? Non certo il Figlio (il quale altro non è che uno smanettone con quattro conoscenze d'accatto mutuate dalle riviste del settore), né tanto meno il terzo membro del consiglio di amministrazione, che è quasi sempre in viaggio. Quindi, chi? Bene: adesso ho una risposta ad entrambi i quesiti e buone ragioni per sospettare che le due figure (quelle del manovratore occulto e dell'autore del codice) coincidano, ma non so se esserne contento o spaventato. Comunque è meglio raccontarvi tutto dall'inizio. Il Cliente decide di realizzare un'applicazione per la gestione dell'Universo (che sarebbe quella attualmente in produzio-

ne), così si mette a tavolino, butta giù l'analisi funzionale, aggiunge alcune regole di business (tipo: l'acqua va sempre in basso, di mamma ce n'è una sola, i negri hanno la musica nel sangue e via dicendo), poi passa all'implementazione. Parte con qualcosa di piccolo e di facile, ma il risultato che ottiene non lo soddisfa; inoltre, come molti creativi, non è particolarmente attratto dalla realizzazione pratica di quello che ha elaborato concettualmente, perciò decide di subappaltare il lavoro a terze parti e le crea contestualmente. I nuovi arrivati studiano l'analisi, definiscono il disegno della base dati, stabiliscono alcuni criteri di implementazione (si passa dal FORTRAN al COBOL) e finalmente cominciano a scrivere il codice. L'analisi, va da sé, è estremamente precisa e tutto andrebbe nel migliore dei modi se a un certo punto uno dei programmatori non cominciasse a inserire delle backdoor nel programma. Inizia quasi per scherzo (modifica il codice in modo da garantirsi l'imbattibilità a briscola e il possesso di Viale dei Giardini e Parco della Vittoria a Monopoli), ma poi ci prende gusto e comincia ad aggiungere modifiche sempre più sostanziali, coinvolgendo nell'operazione di inquinamento anche alcuni suoi colleghi. Alla fine, i suoi privilegi sono di poco inferiori a quelli dell'Amministratore del sistema. Reso spavaldo dall'impunità di cui ha goduto fino ad allora, si mette alla testa del suo gruppo di cialtroni e tenta di esautorare il Cliente con un colpo di mano, ma senza successo: il suo piano è subito scoperto e sia lui che i suoi seguaci sono trasferiti per punizione in una sede secondaria, un open space senza aria condizionata, dove d'estate fa un caldo assurdo. Li salva da un destino peggiore (la presenza, nell'ufficio, di colleghe di sesso femminile) una curiosa circostanza: c'è bisogno di loro per la manutenzione del programma. Infatti, anche se la congiura è sventata, i danni al codice rimangono e non c'è né il tempo di ricontrollare tutto né chi, a parte loro, lo passa fare. Si decide perciò di mettere in produzione l'applicazione così com'è e di procedere all'eliminazione delle anomalie a mano a mano che verranno alla luce: una scelta tanto rischiosa quanto necessaria, che porta ben presto ad una situazione paradossale in cui i congiurati diventano il punto di riferimento del Cliente per la correzione degli errori che loro stessi hanno generato e che seguitano a generare (un po' per vendetta, un po' per accrescere il proprio potere contrattuale) sfruttando i privilegi illegali che si sono garantiti. É facile capire con quanta poca gioia questi personaggi abbiano salutato il mio arrivo: anche se apparentemente masochista, la scelta del Cliente di riscrivere tutta l'applicazione si rivela adesso l'unica possibilità rimasta di eliminare il Male dal Creato. Ecco spiegate le ingerenze esterne, ecco spiegato il motivo. Fatemi gli auguri.

Quarta relazione sul progetto "U++" (1 Maggio)

Credete a me: i clienti buoni esistono solo per i commerciali; per i tecnici esistono solo clienti che non hanno (ancora) creato problemi. Ero così eccitato dall'idea di essere lo Strumento Divino che avrebbe eliminato il Male dall'universo ed invece eccomi qui, sprofondato nella più tetra depressione per colpa del Cliente e delle sue assurde fissazioni. Io pensavo, anzi, ero certo che nella nuova versione dell'applicazione la parola d'ordine sarebbe stata "Bene": con quale sorpresa ho scoperto invece che, anche in questa versione, l'esistenza sarà un'altalena fra le colline della moderata contentezza e gli abissi oceanici della malinconia. Stamattina ne ho parlato con il Cliente, ho cercato di farlo ragionare. Addirittura, visto che erano qui e che non avevano niente da fare, mi sono perfino fatto dare una mano da Schopenhauer e Bergson, ma invano. Mi ha risposto, seccato, di fare il mio lavoro e di non impicciarmi di questioni che vanno oltre la mia comprensione (per poi blandirmi perché lo aiutassi con i backup dopo che aveva cancellato per sbaglio due galassie). Nel pomeriggio sono tornato alla carica, convinto com'ero che le cospicue quantità di vin santo ingurgitate nel post-prandiale avessero favorevolmente influito sulla sua disponibilità, ma mi sbagliavo: mi ha ripetuto quanto aveva già detto nella mattina e, in più, mi ha attaccato un pistolotto di sapore vagamente taoista, il cui fine dichiarato era quello di illuminarmi sulla soggettività dei concetti di bene e di male. In un ultimo, disperato tentativo, ho ribattuto che la presenza del Male nel Creato è uno degli argomenti preferiti di coloro che tentano di dimostrare che Dio non esiste, ma è equivalso a darsi la zappa sui piedi: mi ha confessato di essere stato proprio lui a infondere simili pensieri nella mente dei suoi detrattori nella speranza, frustrata, che la gente si trovasse qualcun' altro da bestemmiare. Una volta di più, mi trovo a dover fronteggiare l'annosa dicotomia fra ciò che il cliente desidera e quello di cui realmente ha bisogno e, una volta di più, sarà ingrato compito dell'uomo informatico quello di venire apparentemente meno ai propri doveri al fine di produrre qualcosa di consono alle reali esigenze del committente.

Fax del Cliente (19 Maggio)

La presente per informarvi della nostra decisione di interrompere l'opera di *porting* del nostro sistema gestionale in linguaggio C++; una decisione sofferta, ma irrevocabile, cui siamo giunti avendo constatato che i principali difetti dell'applicazione precedente non avevano trovato una cura in questa nuova versione. Stando così le cose, abbiamo deciso di focalizzare i nostri sforzi piuttosto che su una rischiosa operazione di revisione, su un programma organico di formazione degli utenti, nella speranza che una maggiore consapevolezza

> delle problematiche di fondo del contesto di utilizzo possa correggere quelle che, a questo punto, consideriamo delle idiosincrasie congenite del sistema. Quale ricompensa dell'impegno da Voi profuso in questo progetto abbiamo pensato di risparmiare l'edificio in cui ha sede la Vostra società dal terremoto del 24 luglio p.v., ma siamo aperti a qualsiasi soluzione di Vostro gradimento, a patto che non contravvenga alle nostre politiche aziendali. Sfortunatamente, il Vostro personale distaccato presso di noi, è rimasto vittima dell'annosa dicotomia fra ciò che gli utenti di un sistema desiderano e ciò di cui realmente hanno bisogno e siamo certi che non avrete nulla in contrario se lo utilizzeremo come cavia per il nostro programma di rieducazione. Attualmente, comunque, gode di ottima salute e sembra non avere difficoltà ad integrarsi con il nuovo gruppo di lavoro cui è stato assegnato, nella sede distaccata del nostro CED.

Finito il corso di riqualificazione, il maestro Canaro non fu più lo stesso. Tornò alla sua società, ma vi rimase solo il tempo di dare le dimissioni. Fondò il nostro Ordine e cominciò a predicare la mistica della programmazione, ottenendo subito un grosso seguito.
I tempi erano maturi per il cambiamento: c'era stata la crescita prodigiosa di fine millennio e c'era stata la rovinosa caduta dell'11 marzo 2000. La vecchia guardia era stanca di dover dipendere dalle bizze della borsa e del marketing; i giovani volevano la Terra Promessa per cui avevano abbandonato gli studi classici; gli utenti volevano che la posta elettronica funzionasse.
La dottrina del maestro Canaro era semplice: riportare l'informatica in una torre d'avorio, accettarne la componente imponderabile ed elevarla da attività produttiva ad attività mistica. Basta con l'improvvisazione, basta con il *time to market*, basta con i prodotti inaffidabili: chi voleva scrivere codice doveva entrare in un Ordine monastico e seguire un lungo percorso di iniziazione, prima di poter cominciare a lavorare.
Furono in molti a vedere nel maestro Canaro *"il vento che spazzerà via il fango"*: la vecchia guardia fu ben felice di veder nuovamente riconosciuto il suo status di casta eletta; i giovani, che cercavano un riparo, lo trovarono all'interno dei monasteri; gli utenti seppero che le loro posta elettronica avrebbe funzionato e furono felici.
Ci fu anche una componente politica, a favorire il successo della dottrina predicata dal maestro Canaro. La società dipendeva ormai totalmente dai computer e i computer dipendevano dagli informatici. Se un giorno avessero deciso di coalizzarsi e di scioperare, avrebbero messo il Paese, se non il Mondo intero, in ginocchio.
Le psicopatologie tipiche degli informatici, la loro incapacità di gestire, se non addirittura di *concepire* una vita sociale, rendevano questa ipotesi molto poco probabile, ma era comunque un rischio troppo grosso per essere ignorato. Al contrario, l'indottrinamento degli informatici, la loro segregazione in una casta con forti componenti mistico-religiose, li avrebbero tenuti lontani dalle lusinghe dei sindacati e avrebbero reso l'ipotesi di uno sciopero improbabile quanto

l'ipotesi di uno sciopero dei sacerdoti.

Gli insegnamenti del maestro Canaro non uscirono mai dall'ambito del software. Non nominò mai l'*hardware*, in nessuno dei suoi scritti; diceva che se un uomo configura un *firewall* è un informatico, se attacca due cavi è un elettricista. Una volta chiesi al maestro Canaro perché non avesse mai dato delle indicazioni sull'affidabilità fisica dei server.
"Se è per questo," mi rispose "non nemmeno prodotto tabelle per il calcolo del cemento armato." Solo anni dopo capii cosa intendesse dire.
Malgrado il suo disinteresse per ciò che lui chiamava: "le cose reali", la rivoluzione culturale provocata dal maestro Canaro ebbe delle conseguenze anche per i produttori di computer. La maggior parte di loro fu lenta a percepire i cambiamenti in atto e pagò duramente la sua disattenzione, gli altri prosperarono.
Anche se il Maestro Canaro non prese mai una posizione definita nella disputa che contrappose i sostenitori della dottrina del *Grande Veicolo* (i *mainframe*) e quelli che erano invece favorevoli al *Piccolo Veicolo* (i PC casalinghi), alcuni alti prelati si dichiararono favorevoli al monoteismo dei *mainframe* e, alla fine, i sostenitori del politeismo furono sconfitti.
Negli uffici e nelle case, i PC vennero sostituiti da terminali e gli edifici ebbero tutti, oltre al garage e al locale caldaie, un'area CED. Il territorio fu suddiviso in diocesi, e ciascuna diocesi fu affidata a un diacono che aveva il compito di sovrintendere al corretto funzionamento dei diversi sistemi.
Sebbene noi, che gli eravamo vicini fin dall'inizio della sua predicazione, sapessimo che era il C++ il suo linguaggio di programmazione preferito, il maestro Canaro lasciò che fosse Java a diventare il linguaggio ufficiale della comunità, subendo senza replicare gli attacchi di coloro che lo accusavano di essersi venduto alla Sun. Le procedure esistenti furono tutte riscritte; gli ordini monastici che insegnavano il C++ divennero sempre di meno e alla fine non ce ne furono più.
Fu solo allora che il maestro Canaro mi parlò per la prima volta del *C'hi++*.

Le nuvole che oscuravano il cielo questa mattina sono scomparse, il cielo della notte trabocca di stelle. Sono cose che si notano, a mano a mano che si invecchia. Io le noto da quando ero bambino.

Quello che ti insegnerò non è illegale, ma è come se lo fosse.

Il buon programmatore

> È stata una giornata estenuante: questa mattina ho tolto una virgola
> dal mio codice e stasera ce l'ho rimessa

Le principali discipline dell'informatica sono: la gestione del personale, la gestione dei progetti, la gestione dei sistemi e la programmazione. Un buon maestro deve essere abile in tutte le discipline, ma può eccellere solo in una di esse, perché per arrivare all'eccellenza dovrà dedicare a quella disciplina tutta la sua vita e nessuno ha più di una vita.
È fondamentale che il maestro conosca e pratichi tutte le discipline, perché solo conoscendo le problematiche connesse a ciascuna fase e a ciascuna disciplina, potrà raggiungere l'eccellenza in una di esse. La disciplina più alta é la programmazione, perché non ha a che fare con entità reali, ma solo con concetti astratti.
Non c'è differenza fra lo scrivere codice e il gestire un'azienda: è come osservare una medesima città su carte di diversa scala. Se il rapporto di scala è basso, si avrà una buona visione di insieme, ma pochi dettagli; se il rapporto di scala è alto, si avranno molti dettagli, ma una visione di insieme limitata. La città, però, sarà sempre la stessa.
Nella *Bhagavad Gita* si dice:

> Tu hai diritto solo all'azione, mai ai suoi frutti: che il tuo movente
> non sia il frutto dell'azione, né vi sia in te attaccamento all'inattività.

È così che opera il buon programmatore: scrive del buon codice, anche se non beneficerà mai dei suoi effetti. Scrive del buon codice perché non può farne a meno.
Scrivere codice non è un mestiere, è una deformazione mentale. Il buon programmatore è un disadattato che ha la fortuna di essere retribuito per dare sfogo ai lati peggiori della sua personalità. Mentre scrive il codice, la mente del buon programmatore lavora su tre livelli: il primo livello decide la sequenza delle istruzioni necessarie a far funzionare il programma; il secondo livello ne controlla costantemente la correttezza sintattica e semantica; il terzo livello verifica se ci sia un sistema più efficiente per fare la stessa cosa. I buoni programmatori sono paranoici ed è giusto che sia così. Le istruzioni di un programma sono eseguite milioni di volte nel corso della loro vita e anche una probabilità minima di errore è inammissibile. Ricòrdati: il codice funziona come vuoi tu solo se non gli lasci altra scelta.
Nei tempi andati ci fu chi sostenne una teoria bizzarra secondo la quale i programmi, per essere veramente efficaci, sarebbero dovuti essere rapidi, ridondanti e imprecisi. L'idea di fondo era che, essendo impossibile, sopra un certo livello di complessità, produrre del codice privo di errori, sarebbe stato meglio produrre molti programmi mediamente imprecisi e valutare l'insieme dei loro risultati. L'unica cosa che posso dire a loro discolpa è che a quei tempi si pensava alla produzione di codice come a un'attività di tipo industriale e non come a una disciplina mistica.

Anche il buon sistemista è paranoico, ma un buon programmatore difficilmente potrà essere un buon sistemista. Il buon sistemista ricerca la ridondanza, perché la ridondanza aumenta l'affidabilità del sistema. Il buon programmatore rifugge la ridondanza, perché la ridondanza diminuisce l'efficienza del codice. Il programmatore è Mercurio, il sistemista è Vulcano.

Un buon programmatore dovrebbe conoscere più di un linguaggio di programmazione. Che si conosca a fondo solo un particolare linguaggio è ammissibile, ma una visione di insieme è sempre necessaria per operare delle scelte. Quelli che al giorno d'oggi si definiscono "programmatori" non sono nient'altro che dei forzati di un determinato linguaggio. Allevati in batteria come i polli, conoscono soltanto quello, lo ritengono il migliore e si interessano degli altri linguaggi solo allo scòpo di evidenziarne le pecche. Questo è improduttivo e pericoloso perché, quando con il passare del tempo, questi "programmatori" diventeranno analisti, gestiranno i progetti e le persone a loro affidate con la stessa miopia, producendo sistemi instabili, costosi e difficili da mantenere. Ciò è immorale.
All'estremo opposto stanno coloro i quali dedicano il loro tempo e le loro energie allo studio dei linguaggi e delle metodologie di progettazione del passato. Sono capaci di citare costrutti in LISP, Ada o Eiffel con grande precisione, ma non hanno mai scritto una riga di codice originale in vita loro. I più abili riescono a modificare i programmi di esempio che trovano nei libri di testo o nei manuali, sostituendo gli algoritmi con altri brani di codice classico, ma senza alcun apporto creativo. Fra questi individui e i veri programmatori c'è la stessa differenza che passa fra un sessuologo e un pornodivo.

Una volta chiesi al Maestro Canaro: «Maestro, cosa facevano i paranoici prima che fosse inventata l'informatica?» Il Maestro ci pensò un po' su, poi rispose: «Ormeggiavano le barche».

I sistemi di numerazione

> Ci sono 10 tipi di persone: quelli che capiscono la numerazione binaria e quelli che non la capiscono

Per dissuadere i pochi ostinati che sono arrivati a leggere fin qui, voglio parlarti dei sistemi di numerazione.

Secondo antiche leggende, la razza umana sarebbe il frutto di un esperimento di ingegneria genetica operato da alieni esadattili arrivati sulla Terra trecentomila anni prima di Cristo. Che sia vero o no, immagina di avere intorno a un tavolo uno di questi alieni, uno dei tuoi confratelli e un uomo che abbia solo un dito.

Se poggiassi una dozzina di uova sul tavolo e chiedessi a ciascuno di loro di numerarle, cosa otterresti? Il tuo amico, che conta in base alle dieci dita delle sue mani, direbbe che le uova sono pari a una volta tutte le sue dita più due. Se esprimessimo questa risposta con potenze del numero delle dita delle mani del tuo confratello, otterremmo che:

```
1x101 + 2x100 = 12
```

L'alieno, abituato a contare in base alle sue dodici dita, direbbe che le uova sono:

```
1x121 + 0x120 = 10
```

Il disgraziato con un dito, che può considerare solo le due possibilità: *un dito/nessun dito*, sarebbe costretto a fare un calcolo più lungo:

```
1x23 + 1x22 + 1x21 + 0x20 = 1100
```

Questi tre risultati, per quanto differenti, non sono sbagliati, perché ciascuno dei tre conta le dodici uova in base al numero delle proprie dita e risponde di conseguenza. Le uova rimangono le stesse: cambia solo il sistema di numerazione.
Del resto, è naturale che sia così: i numeri sono solo concetti astratti; simboli che si utilizzano per identificare delle quantità. Potresti sostituire i numeri da 0 a 9 con le prime dieci lettere dell'alfabeto e non cambierebbe nulla:

0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i	l

```
a + b = b
b + c = d
d * e = bc
```

Il metodo di calcolo resta lo stesso: quando la quantità da valutare è maggiore del numero delle cifre disponibili, si riporta la differenza a sinistra, nella colonna di ordine superiore. Il valore *bc* della moltiplicazione, significa infatti:

```
b*(numero di cifre)1 +
c*(numero di cifre)0
```

ovvero, visto che il numero di cifre a nostra disposizione è 10:

```
b*10 + c*1
```

Se sostituiamo le lettere con i numeri, otteniamo:

```
1*10 + 2*1 = 3*4
```

La numerazione esadecimale fa proprio questo: utilizza le lettere dalla A alla F in aggiunta ai dieci valori della numerazione araba:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Questo torna molto utile quando hai a che fare con dei byte di otto cifre, come vedremo dopo.

Nei sistemi di numerazione *posizionali* il valore di una cifra dipende dal punto in cui si trova: le cifre a sinistra hanno più valore delle cifre a destra, ma non tutti i sistemi di numerazione funzionano allo stesso modo. Se tu chiedessi a un antico romano di calcolare il numero delle uova, ti risponderebbe:

```
XII
```

Anche questo sarebbe un risultato corretto, pur se diverso dai precedenti. Il sistema di numerazione romano è un po' più complicato dei sistemi in base 10, 12 e 2 che abbiamo visto finora. Nella numerazione romana, il valore di una cifra (I, V, X, L, C, D, M) non dipende dalla sua posizione assoluta, ma dalla sua posizione in rapporto alle altre cifre.
Il Maestro Canaro nacque il 29 febbraio 1964; in cifre romane, questa data è:

```
XXIX II MCMLXIV
```

ovvero:

```
10+10+(10-1)

1+1

1000+(1000-100)+50+10+(5-1)
```

I numeri romani possono andar bene per un popolo di conquistatori: sono facili da scolpire nel marmo, a memoria di una battaglia vittoriosa e non considerano lo zero, la nullità, ma per il commercio e per i calcoli in generale sono piuttosto scomodi. Quando gli arabi diffusero nel bacino del Mediterraneo il sistema di numerazione che avevano a loro volta imparato dagli indiani, la sua diffusione fu tanto rapida quanto inevitabile.
I computer, però, non sono né egocentrici come un imperatore romano, né astuti come un mercante fenicio; al contrario, sono veloci, ma tonti: bisogna spiegar loro le cose in maniera semplice, perché le possano portare a termine.
La numerazione romana non fa per loro, perché richiede troppi simboli e troppe regole per essere messa in pratica. La numerazione decimale è già un po' meglio, ma richiede comunque la definizione di dieci simboli diversi, uno per ciascuna cifra da 0 a 9. Paradossalmente, il sistema di calcolo che un calcolatore può capire più facilmente è quello per noi più complicato, ovvero il sistema binario. Questo sistema richiede solo la definizione di due simboli: *1* e *0*; due concetti che possono capire facilmente sia un transistor che una scheda perforata.

Qualcuno potrebbe dirti che ci sono sistemi di calcolo migliori del binario. Qualcuno, probabilmente uno di quei disgraziati individui che amministrano le basi di dati, potrebbe dirti che il sistema ternario bilanciato funziona molto meglio e semplifica i calcoli. Ciò corrisponde al vero.
La ragione per cui il Maestro Canaro non approvava i sistemi ternarii era filosofica, non tecnica. I due valori del sistema binario possono essere identificati con i valori logici sì/no, vero/falso; ovvero con le due condizioni di esistenza e non esistenza, per esempio, di un buco in una scheda perforata o di una corrente in un circuito. Al contrario, i sistemi con basi superiori a due, richiedono la definizione di un terzo stato che non può essere né sì, né no; né vero, né falso; né esistente, né in-esistente. Questo terzo stato intermedio può essere definito solo per mezzo di una valutazione che non è logica, ma quantitativa. Mettiamo che su una scheda perforata il valore 0 sia associato all'assenza di fori e il valore 1 sia associato alla presenza di un foro; un terzo valore, differente da 0 e da 1, potrebbe essere identificato o da un foro di dimensioni differenti o da una concavità. Quale che sia la soluzione scelta, saremmo costretti a definire un valore in base a una grandezza fisica, ovvero, a basare il calcolo digitale su una misurazione analogica.
Il Maestro Canaro pensava che ciò fosse empio.

Un esempio di tutto ciò di cui ti ho parlato oggi lo trovi nella codifica RGB dei colori delle pagine Web. Nella codifica RGB, ciascun colore è definito come una combinazione di rosso verde e blu – i colori *Red*, *Green* e *Blue* le cui iniziali costituiscono l'acronimo del sistema.
La quantità di ciascun colore è determinata da un valore di 8 bit e può quindi variare da 0 a 255 (2^8).
I possibili colori della codifica RGB possono essere calcolati o moltiplicando fra loro i tre valori di R, G e B:

```
256 * 256 * 256 = 16.777.216
```

oppure considerando i tre byte un unico valore binario di 24 cifre:

```
224 = 16.777.216
```

Un colore RGB con tutti e tre i suoi byte a zero è nero; uno con tutti i byte di valore 255 è bianco.

RGB(0,0,0)
RGB(255,255,255)

Fra questi due estremi ci sono le restanti 16.777.214 possibili combinazioni dei tre valori. Se assegnamo un valore solo a uno dei tre byte, otterremo delle gradazioni di rosso, verde o blu:

RGB(234,0,0)
RGB(0,111,0)
RGB(0,0,80)

Se attribuiamo dei valori a tutti e tre i byte, otterremo dei colori composti. Per esempio, il colore di sfondo dei brani di codice del tuo libro o il colore dei link di navigazione:

RGB(232,232,222)
RGB(85,85,0)

Gli stessi valori possono essere espressi in notazione esadecimale:

000000
FFFFFF
ea0000
006f00
000050
e8e8de
555500

L'unica differenza è che la numerazione esadecimale è più facile da utilizzare e da memorizzare, perché ciascuna lettera corrisponde a un byte. Il valore binario e il colore visualizzato, rimangono gli stessi, indipendentemente dal modo in cui tu li vuoi misurare.

C'è qualcosa, in tutto questo, sulla quale vorrei che tu soffermassi la tua attenzione: i numeri *descrivono* un valore, non *sono* un valore. I valori esistevano da prima che esistessero i numeri e gli sopravviveranno. La quantità di fiammiferi che vedi qui sul mio tavolo, rimane la stessa sia che tu la chiami *4*, *100* o *IV*, così come rimarrebbe la stessa se tu la chiamassi *five* o *cinco*.
I numeri li abbiamo inventati noi, i valori, no.

I linguaggi di programmazione

> Dio non programma in Assembler con l'Universo

Brian Kernigan, disse:

> L'unico modo di apprendere un linguaggio di programmazione è quello di utilizzarlo per scrivere programmi.

Il maestro Canaro sosteneva che anche scrivere il manuale di un linguaggio di programmazione è un buon sistema per impararlo.

I linguaggi di programmazione servono a chiedere al computer di fare determinate operazioni. Perché il computer possa ubbidirci, deve "capire" che tipo di entità deve gestire e che tipo di operazioni deve compiere su di esse. Per esempio, se chiedessimo al computer di raddoppiarci lo stipendio, il computer dovrebbe sapere almeno a quanto ammonta il nostro stipendio e come funziona una moltiplicazione per due.

```
long raddoppia(long stipendio)
{
    return stipendio * 2;
}
```

Siccome l'unica cosa che il computer conosce sono delle sequenze di 1 e di 0 – siano esse su un disco, in memoria o sulla porta di comunicazione della tastiera o del video, – dobbiamo trovare il modo di "spiegargli" i concetti di *stipendio* e *moltiplicazione* in forma binaria. I linguaggi di programmazione fanno precisamente questo: traducono le nostre richieste in un linguaggio che il computer può comprendere.

Il linguaggi di programmazione possono essere di *alto livello* o di *basso livello*. I linguaggi di alto livello, come il *C* o il *C++* hanno dei costrutti simili al linguaggio naturale e non sono legati a una determinata architettura hardware. I linguaggi di basso livello, come l'*assembly* hanno dei costrutti simili al codice nativo della macchina e sono, per questo motivo, legati a uno specifico tipo di hardware.
La funzione *raddoppia*, che abbiamo visto prima, tradotta in linguaggio *assembly*, diventa:

```
raddoppia(long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        add     rax, rax
        pop     rbp
        ret
```

I programmi scritti con linguaggi di alto livello, come vedi, sono più facili da scrivere e da correggere, ma sono più lenti da eseguire perché il codice deve essere "tradotto" in una forma comprensibile alla CPU. I programmi scritti con

linguaggi di basso livello sono più complessi da scrivere o da correggere, ma sono molto più veloci da eseguire perché contengono solo le istruzioni strettamente necessarie a svolgere il compito desiderato.

Un computer è come una nave. L'hardware è in basso, come la sala macchine e la CPU è il capo-smacchinista: non vede il mare, non vede il cielo, non sa nemmeno se la nave stia andando a Nord oppure a Sud; riceve le sue istruzioni dall'interfono e si interessa solo solo della velocità e dei consumi. Il sistema operativo è l'equipaggio: è in una posizione intermedia, né in basso né in alto e si prende cura della nave e del carico; può parlare con il Comandante e con il capo macchinista, ma non parla mai con gli armatori. L'interfaccia utente, invece, è il ponte di comando: è il punto più alto della nave, da cui si può vedere tutto; è in contatto diretto con gli armatori, da cui riceve delle direttive su ciò che deve o non deve fare e con l'equipaggio, che lo informa sullo stato della nave.
Ciascun livello del computer ha il suo linguaggio. La CPU di questo computer "ragiona" in quello che si chiama: *linguaggio macchina*, ovvero una lunga serie di valori decimali che indicano sia le operazioni da compiere che le grandezze coinvolte. Il sistema operativo è scritto in C, perché è il linguaggio che si adatta meglio a gestire un computer. L'interfaccia utente, infine, sfrutta dei linguaggi a oggetti come il C++ per creare gli elementi grafici che le consentono di interagire con l'utente.
I linguaggi e i costrutti che si applicano a un livello non funzionano, se si applicano agli altri livelli, perché ciascun livello concepisce e gestisce gli elementi di un programma in maniera diversa. L'interfaccia utente conosce molte caratteristiche del programmatore: vede il suo viso nella Webcam, ascolta la sua voce nel microfono, talvolta legge le sue impronte digitali sullo scanner. Il sistema operativo non lo può vedere o sentire, ma conosce i suoi dati anagrafici e sa tutto quello che lui scrive ai suoi amici. La CPU non sa nulla di lui: gestisce i suoi dati, ma non sa cosa siano. Può elaborare la sequenza di byte:

```
01000011
01100001
01101110
01100001
01110010
01101111
```

ma non sa che è il nome: *Canaro*. In effetti, non sa nemmeno cosa sia, la parola: *nome*. Per lui è solo la sequenza di byte:

```
01001110
01101111
01101101
01100101
```

Invertendo questi concetti, una volta il Maestro Canaro disse:

> L'Uomo non può dire se Dio sia buono o meno, perché non si può classificare l'essenza del Creatore con gli attributi del creato. Sarebbe come se i byte di un computer si chiedessero se il Programmatore valga 1 o 0.

Noi percepiamo il mondo reale come un insieme di eventi che coinvolgono una o più entità. Ciascuna entità è identificabile per le sue caratteristiche fisiche e ha un suo modo particolare di reagire agli stimoli esterni. Per descrivere le entità, i linguaggi di programmazione hanno i tipi di dato; per descrivere gli eventi, istruzioni e operatori Un linguaggio di programmazione è tanto più efficace quanto più i suoi tipi di dato e i suoi costrutti sintattici riescono a rappresentare le entità che il programma dovrà gestire.
Ci sono molti linguaggi di programmazione. Ciascun linguaggio sa spiegare bene certe cose e meno bene certe altre. Esistono linguaggi che descrivono bene le transazioni bancarie, ma che sono inadatti a gestire il traffico aereo; altri che funzionano molto bene per creare pagine Web, ma con cui sarebbe masochistico scrivere un sistema operativo. Un programmatore che conosce a fondo un certo linguaggio di programmazione può utilizzarlo efficacemente per scrivere qualsiasi programma, ma a quale prezzo? Se il linguaggio di programmazione è inadatto a descrivere gli eventi che interesseranno il programma, il programmatore dovrà preoccuparsi di piegare o gli eventi o il linguaggio o entrambi per ottenere il risultato atteso. Scegliendo invece un linguaggio i cui tipi di dato e i cui costrutti siano più affini alla realtà da descrivere, si risparmiano tempo e fatica e si scrive un codice più affidabile.
I linguaggi di programmazione possono essere di due tipi: *interpretati* o *compilati*. Posso spiegarti la differenza in questo modo: immagina di essere in Cina e di dover tornare al tuo albergo in taxi. Se non parli il Cinese, ha due possibilità: o porti con te un interprete che spieghi al tassista dove devi andare o ti fai dare dall'albergo un foglio di carta con l'indirizzo e lo fai vedere al guidatore del taxi. Entrambe le possibilità hanno lati positivi e negativi. Se scegli di portarti dietro un interprete avrai dei costi in più, ma sarai libero di andare dove vuoi: in albergo, in un ristorante o in un locale notturno. Se invece opti per il foglio con l'indirizzo risparmierai i soldi dell'interprete, ma, una volta salito in taxi, potrai solo tornare all'albergo.
Con i linguaggi di programmazioni avviene la stessa cosa. Il codice dei linguaggi interpretati viene letto da un programma chiamato: *interprete* che prima traduce le istruzioni nel linguaggio del computer e poi le esegue. Il codice dei linguaggi compilati, al contrario, viene letto da un programma chiamato *compilatore*, che lo converte in istruzioni comprensibili dal computer, scrive queste istruzioni in un nuovo file, detto: *object-file* e lo passa a un altro programma, chiamato *linker*, che lo trasforma in un file *eseguibile*.
Anche in questo caso, ciascun metodo ha lati positivi e lati negativi.
I linguaggi interpretati sono più lenti, perché il passaggio lettura/conversione/esecuzione si ripete ogni volta che il codice viene elaborato; inoltre, richiedono delle risorse di sistema in più da destinare all'interprete. In compenso, ogni modifica

al codice sorgente viene trasmessa immediatamente anche al programma in esecuzione. I linguaggi compilati sono più rapidi e richiedono meno risorse di sistema perché la compilazione avviene solo una volta, dopo di che il programma può essere eseguito autonomamente, ma se si modifica il codice sorgente, il programma dovrà essere ri-compilato per funzionare correttamente. Anche la religione è, da un certo punto di vista, un linguaggio di programmazione, perché consente agli umani di dialogare con Dio. Il codice, in questo caso, è la preghiera, che i fedeli formulano e inviamo alla Divinità nella speranza che la esegua. Anche le religioni, come i linguaggi di programmazione, possono essere o compilate o interpretate, a seconda che la comunicazione fra il fedele e Dio sia diretta o mediata da un Ordine sacerdotale.

Tutti i linguaggi hanno una sintassi, tutti i linguaggi hanno dei tipi di dato; pochissimi linguaggi hanno anche dei principii. I principii sono importanti, perché danno robustezza al linguaggio. Il linguaggio C, da cui è derivato il C++, non ha principii: è solo un modo per descrivere gli eventi all'interno di un computer. L'istruzione:

```
char b = 23;
```

è un modo per dire al computer:

1. trova un'area di memoria libera larga 1 byte;
2. assegna l'indirizzo di quell'area alla variabile *b*;
3. inserisci nell'area di memoria il valore *23*.

Immagina che tu sia il C. Il muro della cucina è la memoria del computer e ciascuna mattonella è un *bit*. Le prime diciassette mattonelle sono occupate, ma dalla diciottesima alla venticinquesima ci sono otto mattonelle libere, così tu prendi un pennarello e, senza farti vedere dal cuoco, scrivi nelle otto mattonelle libere il corrispondente binario del numero 23:

---

0 0 0 1 0 1 1 1

La variabile *b*, a questo punto, avrà un *indirizzo*, ovvero il numero della prima mattonella in cui hai scritto (18) e un *valore*, ovvero il numero binario contenuto nelle otto mattonelle (23).
Se l'istruzione successiva fosse:

```
char c = b * 2;
```

tutto quello che dovresti fare è di cercare un'altra serie di otto mattonelle libere in cui scrivere il valore della variabile *b* spostato verso sinistra di una mattonella:

0	0	0	1	0	1	1	1	
		0	0	1	0	1	1	1	0

Ti vedo perplesso, ma ragiona: la moltiplicazione per due, in binario, è l'equivalente della moltiplicazione per dieci nella numerazione decimale. Se devi moltiplicare un numero decimale per la base del sistema di numerazione — che è dieci, appunto —, lo sposti a sinistra di un posto e aggiungi uno zero a destra:

```
1234 * 10 = 12340
```

Se devi moltiplicare un valore binario per la base del sistema, che in questo caso è due, fai la stessa cosa:

```
00010111 * 00000010 = 00101110
```

Il valore `00101110` equivale a 46, che è il doppio di 23.

Il C è un linguaggio estremamente efficiente per gestire tutte le entità che puoi trovare in un computer – numeri, stringhe, flussi di dati –, ma la sua forza è allo stesso tempo il suo limite, perché non funziona altrettanto bene se hai a che fare con oggetti più complessi. Anche la gestione delle stringhe di testo, nel C, è rudimentale: se tu provassi a eseguire la somma di cui parla il Maestro Canaro nel suo diario:

```
#include <stdio.h>

int main (int argc, char** argv)
{
    char* a = "pippo";
    char* b = "pluto";
    printf("\npippo+pluto=%s\n", a+b);
    return 0;
}
```

il compilatore C ti avviserebbe che non può sommare due variabili di tipo char.

```
% gcc linguaggi-programmazione-pippopluto.c -o ../out/esempio;
6.1-pippopluto.c:7:35:
error: invalid operands to binary expression
('char *' and 'char *')
    printf("\npippo+pluto=%s\n", a+b);
                                ~^~
1 error generated.
```

Il C è un linguaggio che non va oltre l'ambito per cui è stato creato. È una convenzione. Se domani smettessimo di utilizzare i computer, il C non esisterebbe più, perché tutti i suoi costrutti sono legati al funzionamento degli apparati informatici. Al contrario, il C++ e tutti i linguaggi che si basano sul paradigma a oggetti hanno una componente filosofica che trascende l'ambito informatico. Se domani smettessimo di utilizzare i computer, i costrutti che il C++ ha ereditato dal C non avrebbero più senso, ma i concetti di classe e di oggetto resterebbero ancora validi.
Molti pensano che avere dei principii sia limitativo, ma non è vero. Il C non ha principi e i suoi costrutti sono limitati ai tipi di dato previsti. Il C++, grazie ai suoi principii, ha una maggior duttilità e può evolversi, adattando i suoi costrutti a qualsiasi contesto.
Il Maestro Canaro pensava che la Scienza fosse come il C: uno strumento perfetto per descrivere ciò che ci circonda, ma inadatto a descrivere ciò che trascende la nostra conoscenza diretta. Il problema, secondo lui, era la matematica. Parafrasando Karl Kraus, diceva che la matematica è una malattia di cui pensa di essere la cura. Non so se conosci la frase di André Weil: «Dio esiste perché la matematica è coerente; il Diavolo esiste perché non possiamo dimostrarlo.» È vero il contrario. I paradossi sono la crittografia di Dio e la matematica è utile non perché quasi sempre è coerente, ma perché, di quando in quando, produce dei paradossi.
Le filosofie occidentali cercano di sfuggire il paradosso; lo Zen, al contrario, lo cerca incessantemente. È possibile che sia il paradosso, la chiave per arrivare alla Verità? O anche: è possibile che non si possa arrivare alla Verità, se non si accetta il paradosso? Del resto: perché il mondo di Dio, la sua visione delle cose, non dovrebbe essere paradossale, per noi? Se mostrassimo al personaggio bidimensionale di un gioco per computer il nostro mondo tridimensionale, non lo troverebbe paradossale? E se spiegassimo a *PacMan* che dobbiamo pagare l'IVA, non lo troverebbe paradossale?
Un'altra ipotesi è che il paradosso indichi il punto di confine fra il nostro sistema e un sistema di ordine superiore. Potrebbe essere, cioè, che nel nostro sistema ci siano degli elementi "di frontiera" che segnano il confine fra il nostro sistema e un altro e che per questa loro ambivalenza non possono essere spiegati compiutamente con gli elementi a nostra disposizione. Il Tempo non si può definire senza una tautologia; definire la Verità comporta una ricorsione.. È la nostra corda, che è ancora troppo corta, o è piuttosto il pozzo che non ha fondo, almeno in questo Universo?

Un giorno un discepolo chiese al maestro Canaro: «Maestro, con quale tipo di arco si tira meglio?».
Il maestro rispose: «Con quello che utilizzi tutti i giorni».
Il discepolo chiese allora: «Maestro, questo vuol dire che si può utilizzare uno linguaggio di programmazione inadatto, purché si raggiunga lo scopo?».
Il maestro sorrise benevolmente e disse: «Ti prego, va' a meditare sull'insensatezza delle tue domande mentre pulisci le latrine comuni.»

Per molto tempo non riuscii a capire il comportamento del maestro Canaro. Per un po', credetti che l'errore fosse quel "meglio" nella prima domanda e che il maestro avesse dato una risposta volutamente imprecisa allo scopo di portare alle estreme conclusioni un ragionamento male impostato, ma mi sbagliavo.
La verità era che il maestro Canaro, come mi confessò lui stesso, stava cercando di scaricare su qualcuno il suo turno di pulizia latrine e il giovane e impetuoso discepolo gliene aveva dato occasione.

Il C++

> Non chiederti cosa può fare per te il sistema operativo; chiediti invece cosa puoi fare tu, per il sistema operativo.

In uno dei suoi libri, Bjarne Stroustrup definì il C++:

> Un linguaggio di programmazione per svolgere compiti non banali.

Fà che si possa dire la stessa cosa di te.

C++ è un linguaggio di programmazione creato da Bjarne Stroustrup nel 1983, quando lavorava ai Laboratori Bell della AT&T. Dieci anni prima, il suo collega Dennis Ritchie aveva creato un linguaggio di programmazione chiamato *C*. Il C, come ti ho detto, era estremamente efficace se dovevi programmare i computer, ma — così come alcuni dei tuoi confratelli — non gestiva altrettanto bene le entità della vita reale.
Stroustrup, che ai tempi del suo dottorato aveva lavorato con un linguaggio a oggetti chiamato *Simula67*, pensò che se avesse potuto aggiungere alla velocità di esecuzione del C la possibilità di creare dei nuovi tipi di dato di Simula, avrebbe ottenuto il linguaggio perfetto. Aveva ragione.
Il nome *C++* si riferisce all'operatore `++`, che serve a incrementare di un'unità il valore di una variabile:

```
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    int C = 12;
    cout << "Valore di C = " << C << endl;
    C++;
```

```
    cout << "Valore di C = " << C << endl;
    return 0;
}
```

Se compili ed eseguti questo codice, otterrai:

```
> g++ cplusplus-incremento.cpp -o ../out/esempio
> ../out/esempio
Valore di C = 12
Valore di C = 13
```

C++, infatti, non era un nuovo linguaggio: era un C migliorato. Tutto il codice e l'esperienza che erano state fatte fino ad allora sul C potevano essere applicate anche al C++.
Parafrasando Neruda, Stroustrsup fece con il C quello che Gesù fece con l'Ebraismo: così come il Nazareno prese una religione spartana, adatta per un popolo in fuga nel deserto e la migliorò, rendendola meno autoritaria, il Danese trasformò un linguaggio pensato per gestire unicamente i computer, in un linguaggio che poteva gestire ogni cosa.

Le principali novità aggiunte al C dal C++ sono: l'*astrazione dei dati*, la *programmazione a oggetti* e la *generic programming*. Ora ti spiego cosa sono, ma tu non preoccuparti se non capisci: approfondiremo tutti questi concetti in seguito.
I tipi di dato del C sono:

```
char, int, float, double, array, enum, struct, union
```

Più che sufficienti per scrivere il *kernel* di Unix, ma decisamente inadeguati per scrivere un programma che gestisca delle linee telefoniche. Perché un linguaggio di programmazione possa gestire con la stessa facilità un flusso di dati, un utente, una linea telefonica o anche un allevamento di cavalli, è necessario che oltre ai tipi di dato predefiniti, possa gestire anche delle nuove entità definite dal programmatore. Questa è ciò che si chiama *data abstraction* e il C++ la ottiene per mezzo delle *classi*. Le classi sono la rappresentazione, all'interno del codice, di un'entità:

```
class Cavallo {
private:
    Sesso  _sesso;
    string _razza;
public:
    const char* getSpecie() const {
        return "cavallo";
    }
    const char getSesso() const {
        return (char)_sesso;
    }
    const char* getRazza() const {
```

```
        return _razza.c_str();
    }
    Cavallo() {}
    Cavallo(const char* razza, Sesso sesso ) {
        _razza  = razza;
        _sesso  = sesso;
    }
};
```

o di un concetto:

```
class Monta {
private:
    Cavallo _maschio;
    Cavallo _femmina;
    Data     _giorno;
public:
    Monta(Cavallo& maschio, Cavallo& femmina) {
        _maschio = maschio;
        _femmina = femmina;
        time(&_giorno);
    }
};
```

Grazie alle classi, il programmatore può creare dei nuovi tipi di dato e utilizzarli all'interno del suo programma nello stesso modo in cui utilizzerebbe i tipi di dato primitivi del linguaggio.
Ciascuna classe ha degli *attributi* e dei *metodi*. Gli *attributi* sono dei dati che descrivono le caratteristiche della classe, per esempio, la razza o il sesso di un cavallo. I *metodi* sono delle funzioni che definiscono il modo in cui la classe può interagire con gli altri elementi del programma. Nelle classi dell'esempio gli *attributi* sono gli elementi che vedi nella sezione `private`, mentre i *metodi* sono le funzioni che vedi nella sezione `public`. La funzione che ha lo stesso nome della classe si chiama *costruttore* della classe, perché "spiega" al compilatore come debbano essere creati gli oggetti di questa classe.
Le classi, però, sono la ricetta, non sono la pietanza. Per poter essere utilizzate, le classi devono essere *istanziate* negli oggetti:

```
int main()
{
    Cavallo stallone("lipizzano", maschio);
    Cavallo giumenta("maremmano", femmina);
    Monta   monta(stallone, giumenta);
    cout << monta << endl;
    return 0;
}
```

`stallone`, `giumenta` e `monta` sono tre oggetti. I primi due sono istanze della

classe `Cavallo`, il terzo è un'istanza della classe `Monta`.
Se aggiungi un po' di codice alle classi che abbiamo visto prima e compili il programma, otterrai:

```
% g++ cplusplus-classe.cpp -o ../out/esempio
% ../out/esempio
DATA: Sun Apr  5 10:38:31 2020
MASCHIO: Specie:cavallo, Sesso:m, Razza:lipizzano
FEMMINA: Specie:cavallo, Sesso:f, Razza:maremmano```
```

Perché un linguaggio di programmazione possa dirsi *orientato agli oggetti*, però, oltre alle classi deve poter gestire l'*ereditarietà* e il *polimorfismo*. L'*ereditarietà* permette di definire dei nuove classi come estensione di classi esistenti:

```
class Animale {
private:
    string _specie;
    string _razza;
    Sesso  _sesso;
public:
    const char* getSpecie() const {
        return _specie.c_str();
    }
    const char getSesso() const {
        return (char)_sesso;
    }
    const char* getRazza() const {
        return _razza.c_str();
    }
    Animale() {}
    Animale(const char* specie, const char* razza, const Sesso sesso ) {
        _specie = specie;
        _razza  = razza;
        _sesso  = sesso;
    }
};

class Cavallo : public Animale {
public:
    Cavallo() {}
    Cavallo(const char* razza, const Sesso sesso )
    : Animale("Cavallo", razza, sesso ) {
    }
};
```

Nell'esempio qui sopra, abbiamo prima definito una classe `Animale`, che ha tre attributi: la `specie` , la `razza` e il `sesso` ; poi abbiamo definito una classe `Cavallo`, derivandola dalla classe `Animale`. In questo modo, se oltre ai cavalli il

nostro programma dovesse gestire anche altri ungulati, non dovremmo ripetere in ciascuna classe le stesse istruzioni, ma potremmo utilizzare quelle della classe base:

```
class Asino : public Animale {
public:
    Asino() {}
    Asino(const char* razza, const Sesso sesso )
    : Animale("Asino", razza, sesso ) {
    }
};
```

A questo punto, la tua sagacia dovrebbe averti fatto rilevare un possibile problema (posto che tu sia sveglio, cosa di cui non sono del tutto certo): la classe `Monta` può gestire solo oggetti di tipo `Cavallo`.
Un linguaggio che non gestisca il polimorfismo ci costringerebbe a scrivere due nuove classi: una per i muli e una per i bardotti:

```
class MontaMulo {
private:
    Asino   _maschio;
    Cavallo _femmina;
    Data     _giorno;
public:
    MontaMulo(Asino& maschio, Cavallo& femmina) {
        _maschio = maschio;
        _femmina = femmina;
        time(&_giorno);
    }
};

class MontaBardotto {
private:
    Cavallo _maschio;
    Asino   _femmina;
    Data     _giorno;
public:
    Monta(const Cavallo& maschio, const Asino& femmina) {
        _maschio = maschio;
        _femmina = femmina;
        time(&_giorno);
    }
};
```

Il problema è che più codice scrivi, più è probabile che farai degli errori e meno facile sarà correggerli. Al contrario, i programmi con meno righe di codice sono più affidabili e più facili da correggere o da modificare.
Il C++ ci aiuta in questo senso perché permette il *polimorfismo*, ovvero la

capacità di una funzione o di un operatore di svolgere il proprio compito indipendentemente dal tipo di dato che deve gestire. Se riscriviamo la classe `Monta` usando, al posto dei parametri di tipo `Cavallo`, dei parametri che hanno il tipo della classe base `Animale`:

```
class Monta {
private:
    Animale* _maschio;
    Animale* _femmina;
    Data     _giorno;
public:
    Monta(Animale* maschio, Animale* femmina) {
        _maschio = maschio;
        _femmina = femmina;
        time(&_giorno);
    }
};
```

Potremo creare degli oggetti di classe `Monta` con qualunque classe derivata:

```
/**
 * @file src/cplusplus-polimorfismo.cpp
 * Esempio di polimorfismo delle classi.
 */

#include <iostream>
#include <ctime>

using namespace std;

typedef time_t Data;

typedef enum _sesso {
    maschio = 'm',
    femmina = 'f'
} Sesso;

class Animale {
private:
    string _razza;
    Sesso  _sesso;
public:
    virtual const char* getSpecie() const {
        return "";
    }
    const char getSesso() const {
        return (char)_sesso;
```

```
    }
    const char* getRazza() const {
        return _razza.c_str();
    }
    Animale() {}
    Animale(const char* razza, const Sesso sesso ) {
        _razza  = razza;
        _sesso  = sesso;
    }
};

ostream& operator << (ostream& os, const Animale& animale) {
    os  << "Specie:" << animale.getSpecie() << "\t"
        << "Razza:"  << animale.getRazza()  << "\t"
        << "Sesso:"  << animale.getSesso()  << endl;
    return os;
}

class Cavallo : public Animale {
public:
    const char* getSpecie() const {
        return "Cavallo";
    }
    Cavallo() {}
    Cavallo(const char* razza, const Sesso sesso )
    : Animale(razza, sesso ) {
    }
};

class Asino : public Animale {
public:
    const char* getSpecie() const {
        return "Asino";
    }
    Asino() {}
    Asino(const char* razza, const Sesso sesso )
    : Animale(razza, sesso ) {
    }
};

class Monta {
private:
    Animale* _maschio;
    Animale* _femmina;
    Data     _giorno;
public:
```

```
    Monta(Animale* maschio, Animale* femmina) {
        _maschio = maschio;
        _femmina = femmina;
        time(&_giorno);
    }
    friend ostream& operator << (ostream& os, const Monta& copula) {
        os << "DATA: "    << asctime(localtime(&copula._giorno))
           << "MASCHIO: " << *copula._maschio
           << "FEMMINA: " << *copula._femmina;
           return os;
     };
};

int main()
{
    Animale* cavallo  = new Cavallo("lipizzano", maschio);
    Animale* giumenta = new Cavallo("maremmano", femmina);
    Animale* asino    = new Asino("amiatino", maschio);
    Animale* asina    = new Asino("sardo", femmina);

    Monta puledro(cavallo, giumenta);
    Monta ciuco(asino, asina);
    Monta mulo(asino, giumenta);
    Monta bardotto(asina, cavallo);

    cout << "PULEDRO\n"  << puledro  << endl;
    cout << "CIUCO\n"    << ciuco    << endl;
    cout << "MULO\n"     << mulo     << endl;
    cout << "BARDOTTO\n" << bardotto << endl;

    return 0;
}
```

Compilando ed eseguendo il programma, otterrai:

```
% g++ cplusplus-polimorfismo.cpp -o ../out/esempio
% ../out/esempio
PULEDRO
DATA: Sun Apr  5 12:33:45 2020
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f

CIUCO
DATA: Sun Apr  5 12:33:45 2020
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Asino   Razza:sardo     Sesso:f
```

```
MULO
DATA: Sun Apr  5 12:33:45 2020
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f

BARDOTTO
DATA: Sun Apr  5 12:33:45 2020
MASCHIO: Specie:Asino   Razza:sardo     Sesso:f
FEMMINA: Specie:Cavallo Razza:lipizzano Sesso:m
```

Come ti ho detto, però, un buon programmatore non si accontenta di essere riuscito a produrre il risultato atteso, ma si chiede sempre se ci sia un modo più efficiente per ottenerlo. Nel nostro caso, il codice che abbiamo utilizzato per mostrare il risultato degli accoppiamenti:

```
Monta puledro(cavallo, giumenta);
Monta ciuco(asino, asina);
Monta mulo(asino, giumenta);
Monta bardotto(asina, cavallo);

cout << "PULEDRO\n"  << puledro  << endl;
cout << "CIUCO\n"    << ciuco    << endl;
cout << "MULO\n"     << mulo     << endl;
cout << "BARDOTTO\n" << bardotto << endl;
```

non è il massimo dell'efficienza, sia perché potremmo sbagliarci ad accoppiare la specie dei genitori con il nome del figlio, sia perché le istruzioni devono essere ripetute per ciascun oggetto. Per risolvere il primo difetto possiamo aggiungere alla classe `Monta` un attributo e un metodo per definire autonomamente che tipo di genìa venga prodotta dalla copula:

```
string   _esito;
void setEsito() {
    if(strcmp(_maschio->getSpecie(),"Asino") == 0) {
        if(strcmp(_femmina->getSpecie(),"Asino") == 0){
            _esito = "asino";
        } else {
            _esito = "mulo";
        }
    } else {
        if(strcmp(_femmina->getSpecie(),"Cavallo") == 0) {
            _esito = "puledro";
        } else {
            _esito = "bardotto";
        }
    }
}
```

ma anche così dovremo comunque riscrivere quattro righe di codice per modificare l'output del programma: un approccio inaccettabile per i sistemi di produzione, dove le entità da gestire possono essere migliaia. Possiamo risolvere questo problema grazie alla *generic programming* e al modo in cui viene implementata nel C++: le classi *template*:

```
template < class T> class list;
```

La classe `list` è una delle classi *template* del C++ e permette di inserire, rimuovere, spostare, unire, ordinare ed elencare una lista di oggetti di una stessa classe. La sintassi per creare una lista di oggetti di classe `Monta` è:

```
list<Monta> monte;
```

Fatto ciò, possiamo aggiungere elementi alla nostra lista con la funzione `push_back()`, alla quale passeremo direttamente il costruttore della classe:

```
monte.push_back(Monta(cavallo, giumenta));
monte.push_back(Monta(asino, asina));
monte.push_back(Monta(asino, giumenta));
monte.push_back(Monta(cavallo, asina));
```

Questo codice è una forma più efficiente di:

```
Monta m1(cavallo, giumenta);
Monta m2(asino, asina);
Monta m3(asino, giumenta);
Monta m4(cavallo, asina);

monte.push_back(m1);
monte.push_back(m2);
monte.push_back(m3);
monte.push_back(m4);
```

Per visualizzare il contenuto della lista, indipendentemente dal numero di elementi, basta l'istruzione:

```
for (list<Monta>::iterator it=monte.begin(); it!=monte.end(); it++) {
    cout << *it << endl;
}
```

La funzione `main()` del nostro programma sarà quindi:

```
int main()
{
    Animale* cavallo  = new Cavallo("lipizzano", maschio);
    Animale* giumenta = new Cavallo("maremmano", femmina);
    Animale* asino    = new Asino("amiatino", maschio);
    Animale* asina    = new Asino("sardo", femmina);
    list<Monta> monte;
    monte.push_back(Monta(cavallo, giumenta));
```

```
    monte.push_back(Monta (asino, asina));
    monte.push_back(Monta (asino, giumenta));
    monte.push_back(Monta (cavallo, asina));
    for (list<Monta>::iterator it=monte.begin();
         it!=monte.end(); it++) {
        cout << *it << endl;
    }
    return 0;
}
```

e l'output che otterremo è:

```
$ g++ cplusplus-template.cpp -o ../out/esempio
% ../out/esempio
DATA:    Sun Apr  5 16:19:24 2020
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f
ESITO:   puledro

DATA:    Sun Apr  5 16:19:24 2020
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Asino   Razza:sardo     Sesso:f
ESITO:   asino

DATA:    Sun Apr  5 16:19:24 2020
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f
ESITO:   mulo

DATA:    Sun Apr  5 16:19:24 2020
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Asino   Razza:sardo     Sesso:f
ESITO:   bardotto
```

Se per qualche motivo volessimo invertire l'ordine degli elementi nella lista, tutto quello che dovremmo fare è di aggiungere prima del ciclo `for` l'istruzione:

```
monte.reverse();
```

e l'output che otterremo è:

```
% g++ cplusplus-template.cpp -o ../out/esempio
% ../out/esempio
DATA:    Sun Apr  5 17:08:27 2020
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Asino   Razza:sardo Sesso:f
ESITO:   bardotto

DATA:    Sun Apr  5 17:08:27 2020
```

```
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f
ESITO:   mulo

DATA:    Sun Apr  5 17:08:27 2020
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Asino   Razza:sardo     Sesso:f
ESITO:   asino

DATA:    Sun Apr  5 17:08:27 2020
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f
ESITO:   puledro
```

Oltre alle classi template predefinite della *Standard Template Library*, il C++ permette di definire le proprie classi template, ma di questo parleremo a tempo debito.
Queste caratteristiche, unite alla compatibilità con il codice scritto in C, fecero di C++ il linguaggio *object-oriented* più utilizzato degli anni '90. L'avvento, alla fine del Secolo, del linguaggio con la "J", quello che ha bisogno di un sistema di *garbage collecion* per sopperire alla pochezza dei suoi programmatori, avrebbe dovuto darci un'idea di quello che sarebbe stato il millennio che ci si presentava davanti. Non a caso, Stroustrsup disse:

> I suspect that the root of many of the differences between C/C++ and Java is that AT&T is primarily a user (a consumer) of computers, languages, and tools, whereas Sun is primarily a vendor of such things.

Nella migliore delle ipotesi, si tratta di codice inefficiente, che esegue le azioni attese, ma richiede sempre più risorse di sistema e/o di tempo di esecuzione. Questo va bene a chi vende l'hardware, perché implica un continuo rinnovo del parco HW con sistemi più potenti. Nella peggiore delle ipotesi, si tratta di codice fallato, che fa male il suo lavoro (così come chi lo ha scritto) e complica la vita e la gestione del sistema.
Non importa a nessuno: l'unica cosa a cui si presta importanza è il rispetto dei tempi di consegna, senza tenere conto del fatto che, spesso, l'utilizzo maldestro di software maldestro allunga i tempi di sviluppo, test e correzione.
In pratica, la programmazione è diventata una sorta di roulette russa, in cui la fortuna gioca un ruolo decisivo: se ti dice culo, gli errori non escono durante il collaudo e se ne riparla in assistenza; se ti dice male, butti la tua vita cercando di far funzionare uno strumento difettoso, invece di spendere un po' di tempo per realizzarne uno sano.
La stessa cosa si può dire delle vite di queste persone. –>

I commenti

> Il Compilatore e il Linker non usano carità, tengono i diecimila oggetti per cani di paglia

I commenti sono la cosa più importante, quando si programma.

Un commento è un breve brano di testo che descrive in linguaggio comune il funzionamento o lo scòpo del codice a cui è riferito:

```
/**
 * Funzione per il raddoppio dello stipendio.
 * Richiede come parametro il valore dello
 * stipendio corrente e ne torna il valore
 * duplicato.
 */
long raddoppia(long stipendio)
{
    return stipendio * 2;   //  raddoppia il valore
}
```

Pur essendo all'interno del codice, il commento viene ignorato in fase di compilazione, perché è un'aggiunta utile solo al programmatore. Al processore non interessa sapere a cosa serve il codice che sta eseguendo: lo esegue e basta, senza farsi distrarre da implicazioni funzionali — o, peggio, etiche — che allungherebbero i tempi di risposta.
È importante commentare bene il proprio codice. Il buon codice produce programmi che durano nel tempo e, col tempo, si tende a dimenticare. Anche se si ha la fortuna di possedere una buona memoria, bisogna considerare la possibilità che il proprio codice sia utilizzato da un collega. Non solo non è educato costringerlo ad analizzare ogni singola riga di codice per capirne il funzionamento, ma è anche improduttivo.
Così come la penuria, anche l'eccesso di commenti è un errore da evitare. Il programmatore inesperto non commenta il proprio codice perché pensa che sia una perdita di tempo. È davvero un atteggiamento poco responsabile. Il neofita zelante riempie il proprio codice di commenti, o perché ritiene che chi lo andrà a leggere sarà meno esperto di lui o perché vuole in questo modo esaltare le sue poche conoscenze. Il programmatore esperto commenta solo ciò che non è di per sé evidente e indica i possibili punti deboli del suo programma in modo che chi si trova a riutilizzarlo o a correggerlo vi presti la dovuta attenzione. Il programmatore perfetto non commenta il suo codice, perché il suo codice è auto-esplicativo:

```
typedef Importo unsigned long;

Importo raddoppiaStipendio(Importo stipendioCorrente)
{
    return stipendioCorrente * 2;
```

```
}
```

È lo stesso brano di codice che abbiamo visto prima, ma stavolta i commenti non servono, perché gli intenti della funzione sono espressi direttamente nel codice, rendendo espliciti il nome della funzione, del suo parametro e del suo valore di ritorno. L'istruzione `typedef`, che hai visto anche negli esempii precedenti, serve a definire dei nuovi tipi di dato, come alias di tipi di dato esistenti. L'istruzione:

```
typedef time_t Data;
```

definisce il tipo di dato `Data` come alias del tipo di dato standard `time_t`, mentre l'istruzione:

```
typedef Importo unsigned long;
```

definisce il tipo di dato `Importo` come alias del tipo di dato standard `unsigned long`.
La definizione dei proprii tipi di dato non solo rende il codice più leggibile, ma lo rende anche più facile da modificare. La funzione originale utilizza dei valori di tipo `unsigned long`, ovvero delle sequenze di 4 byte che permettono di memorizzare numeri interi da 0 a 4.294.967.295 (2^{32}). Questo tipo di dato può essere utilizzato per uno stipendio che non abbia cifre decimali, come quello del Maestro Canaro nel secolo scorso, ma se dovessimo raddoppiare uno stipendio con delle cifre decimali, dovremmo utilizzare dei tipi di dato come i `float` o i `double` e, senza una `typedef`, saremmo costretti a modificare sia il valore di ritorno che il tipo di parametro della funzione:

```
float raddoppia(float stipendio)
{
    return stipendio * 2;   //  raddoppia il valore
}
```

Se invece abbiamo definito un nostro tipo di dato dobbiamo modificare solo l'istruzione `typedef`, lasciando tutto il resto invariato:

```
typedef Importo float;

Importo raddoppiaStipendio(Importo stipendioCorrente)
{
    return stipendioCorrente * 2;
}
```

È ammissibile che i commenti abbiano una sfumatura umoristica, può essere utile per scaricare un po' della tensione in chi legge, così come scambiarsi una battuta mentre si lavora, ma anche in questo non si deve esagerare.

```
    case STATE_CR:  // Got CR: look for LF
        state = STATE_NORMAL;
        if( ch == LF) {
            continue;
        }
```

```
        break;
    default:  // Hmmm... this shouldn't happen
        break;
    }
    cgi.param_data[j++] = ch;
}
```

Il C++ prevede due modi distinti per commentare il codice: due caratteri `//` affiancati o la sequenza `/* */`, che il Linguaggio ha mutuato dal suo predecessore, il C. Nel primo caso, il compilatore ignora i caratteri dal simbolo fino alla fine della riga; nel secondo caso, ignora tutto ciò che è compreso fra le due sequenze di caratteri.

```
/**
 * Questa sintassi è più comoda per i commenti
 * che si estendono su più di una riga
 */
for (int i = 0; i < 10; i++) {
    cout << i << endl; // questa, per i commenti su una riga
}
```

Entrambe le soluzioni hanno dei pro e dei contro che con l'esperienza risultano evidenti. Scegliere l'uno o l'altro è spesso una questione di convenienza, altre volte una questione di stile, ma se si capisce la vera essenza del Linguaggio, è solo una questione di ritmo.

I commenti sono un po' come la letteratura: se si scrive troppo poco è male; se si scrive troppo, dilungandosi in descrizioni inutili, è altrettanto sbagliato. Il paragone vale anche all'inverso: i libri, le opere letterarie, sono i commenti al codice della vita. Si scrive ciò che si desidera ricordare.
Il Maestro Canaro diceva che la cosa più complicata, dello scrivere un manuale, è che, alle volte, per spiegare un determinato concetto, hai bisogno di spiegare prima un altro concetto, che però non può essere compreso se non si capisce il primo. Questo è uno di quei momenti. Il corrispettivo dei commenti, nel C'hi++ sono quelli che il Maestro Canaro chiamava i *Post-It*, ovvero una sorta di memoria di massa dell'Universo in cui è salvato il ricordo delle scelte fatte dagli esseri senzienti in ogni ciclo di esistenza. Io, però, non posso parlarti adesso dei *Post-It*, perché prima dovrei prima spiegarti, se non altro, cosa sono i "cicli di esistenza" e, per il momento, è prematuro farlo. Rimando perciò la trattazione dei *Post-It* a un secondo momento e ti racconto invece un aneddoto sul Maestro Canaro che riguarda proprio i commenti al codice.
Una volta, il maestro Canaro mostrò ai suoi allievi un brano di codice e gli chiese di spiegare che cosa facesse:

```
/************************************************************

#include <iostream.h>
```

```
#define ERR_NOFILE 1

int errore( int errore );
void leggi_file( char * nomefile );
void crea_matrice(void);
int inizializza(void);
void imposta_probabilita(void);
void attrattore(void);
int chiudi(void);

int main(int argc, char * argv[])
{
    int esito = 0 ;
    if(argc < 2) {
        esito = errore(ERR_NOFILE);
    } else {
        leggi_file(argv[1]);
        crea_matrice();
        if(inizializza()) {
            const char * valore = imposta_probabilita();
            attrattore(valore);
            esito = chiudi();
        }
    }
    return esito ;
}

************************************************************/
```

Gli allievi studiarono il codice e diedero le loro risposte, ma sbagliarono tutti, perché tutte le istruzioni erano comprese fra un `/*` e un `*/` e non venivano compilate.

I tipi di dato

> Tutto è byte

La varietà dei tipi di dato disponibili è la caratteristica principale del C++.

Il C++ ha ereditato dal C una vasta gamma di tipi di dato. A seconda della loro natura, possiamo dividere questi tipi di dato in due gruppi: i dati *scalari* e i dati *aggregati*.

I tipi di dato *scalari* sono: `void`, `bool`, `char`, `wchar_t`, `int`, `float`, `double` e permettono di gestire gli elementi classici di un programma: i numeri interi, i numeri decimali, le lettere e i valori booleani.
I tipi di dato *aggregati* sono gli *array*, le *unioni*, gli *emumerati*, le *strutture* e le *classi*. Questi, come dice anche il loro nome, sono costituiti dall'aggregazione di più dati scalari dello stesso tipo o di tipi differenti.
Il prossimo brano di codice — fastidiosamente didascalico, nella sua sostanziale inutilità — mostra i principali tipi di dato del C++. Contiene molti elementi di cui ancora non ti ho parlato, quindi non preoccuparti se non capisci del tutto ciò che fa.

```
#include <iostream>

using namespace std;

/** Definisce un tipo di dato enumerato di nome RGB */
enum RGB { red = 0xFF0000, green = 0x00ff00, blue = 0x0000ff };

/** Definisce una struttura che contiene un colore RGB e un nome */
struct ColoreRGB {
    RGB  valore;
    const char* nome;
};

/**
 * Definisce la classe Colore, che contiene un coloreRGB
 * e una funzione che ne mostra il nome.
 */
class Colore {
public:
    ColoreRGB coloreRgb;
    void nome_colore() {
        cout << "coloreRGB:";
        switch(coloreRgb.valore) {
            case red  : cout << "red"  ; break;
            case green: cout << "green"; break;
            case blue : cout << "blue" ; break;
        }
        cout << endl;
    }
};

int main()
{
    /** Dichiara una serie di variabili */
    bool    booleano  = false;
```

```
    char    carattere = 'C';
    int     intero    = 1234567890;
    float   decimale  = 3.14;
    char    array[]   = "abcdefghilmnopqrstuvz";
    RGB     enumerato = green;

    /** Crea un oggetto di tipo Colore */
    Colore  colore;

    /**
     * Assegna un valore ai dati della struttura coloreRgb
     * all'interno dell'oggetto di tipo Colore.
     */
    colore.coloreRgb.valore = enumerato;
    colore.coloreRgb.nome = "verde";

    /** Mostra il valore delle variabili */
    cout << "booleano:"    << booleano   << endl;
    cout << "carattere:"   << carattere  << endl;
    cout << "intero:"      << intero     << endl;
    cout << "decimale:"    << decimale   << endl;
    cout << "array:"       << array      << endl;

    /** Mostra il nome del colore */
    colore.nome_colore();

    return 0;
}
```

Compilando ed eseguendo questo codice, otterrai:

```
% g++ src/cpp/tipi-di-dato-principali.cpp -o src/out/esempio
% src/out/esempio
booleano:0
carattere:C
intero:1234567890
decimale:3.14
array:abcdefghilmnopqrstuvz
coloreRGB:green
```

Il tipo di dato `void` non ha né una dimensione né un valore e può essere assegnato solo a un puntatore o a una funzione che non torna alcun tipo di valore, come la funzione `nome_colore` della classe `Colore`.
Il tipo di dato `bool` è utilizzato per gestire i valori booleani. Può assumere solo due valori: `true` o `false`, quindi ha la dimensione minima possibile, ovvero un byte.
Il tipo `int` permette di gestire i numeri interi. Di solito ha una dimensione di

quattro byte, che può essere aumentata o diminuita per mezzo di parole-chiave dette: modificatori.
I tipi `float` e `double` si utilizzano per gestire i numeri decimali. Hanno una lunghezza di quattro e otto byte rispettivamente e una precisione di sette e quindici cifre decimali.
I tre elementi che compaiono al di fuori della funzione `main` sono tipi di dato *aggregati* e permettono di definire dei tipi di dati non standard.
Gli `enum` permettono di assegnare dei nomi a dei valori, rendendo più facile la programmazione. Nell'esempio, l'*enum* `RGB` assegna un nome a tre valori della codifica RGB. Questo nuovo tipo di dato è utilizzato nella struttura `ColoreRGB`.
Le `struct` permettono di creare degli insiemi di dati eterogenei. La variabile `ColoreRGB` contiene due variabili differenti: un valore di tipo `RGB` e un puntatore a carattere.
Tutti i tipi di dati che abbiamo visto finora sono un retaggio del linguaggio *C*.
La classe `Colore`, invece, è una delle novità introdotte dal C++ e contiene due elementi: una struttura di tipo `ColoreRGB` e una funzione che mostra il nome del colore.
Attenzione, però: nessuno di questi signori è in realtà ciò che afferma di essere. Nella variabile `booleano` non c'è il valore `true` o `false`, ma un valore binario che il sistema considera tale. Allo stesso modo, nella variabile `carattere`, non c'è la lettera `C`, ma il valore binario corrispondente alla codifica ASCII della lettera `C`. Ogni tipo di dato non è altro che il nome di una quantità di memoria. Ricordatelo, perché questo è la chiave di tutto.

La dimensione di memoria associata a ciascun tipo di dato non è fissa, ma può variare a seconda del sistema in cui stai lavorando. Il codice qui sotto ti permette di conoscere la dimensione in byte dei tipi di dato scalari:

```
/**
 * @file tipi-di-dato-dimensione.cpp
 * Mostra la dimensione dei principali tipi di dato del C++.
 */

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int w = 12;

    cout << setw(w) << "bool: "   << sizeof(bool)   << endl;
    cout << setw(w) << "char: "   << sizeof(char)   << endl;
    cout << setw(w) << "int: "    << sizeof(int)    << endl;
```

```
    cout << setw(w) << "float: "  << sizeof(float)  << endl;
    cout << setw(w) << "double: " << sizeof(double) << endl;

    return 0;
}
```

Compilando ed eseguendo questo codice, otterrai:

```
% g++ tipi-di-dato-dimensione.cpp -o ../out/esempio
% ../out/esempio
      bool: 1
      char: 1
       int: 4
     float: 4
    double: 8
```

Dalla dimensione del tipo di dato dipende la quantità di valori che può assumere. Un `char` è composto da otto bit e può assumere per ciò 2^8 valori, quindi: da 0 a 255. Un `int` è composto da trentadue bit e può assumere 2^{32} valori, ovvero: da 0 a 4.294.967.295. Un `float` ha 8 bit per l'esponente e 23 per la mantissa, quindi può assumere valori da 1.175494e-38 a 3.402823e+38 e così via.

Oltre ai tipi di dato, il C++ ha ereditato dal C anche i cosiddetti *modificatori*, ovvero delle parole-chiave che, se aggiunte al nome di un tipo di dato, ne modificano le caratteristiche.

I modificatori `short` e `long` alterano la dimensione del dato, riducendolo o aumentandolo.

I modificatori `signed` e `unsigned` fanno sì che il primo bit del dato sia considerato il suo segno: positivo se il valore è 0; negativo se il valore è 1. Il segno, però, "ruba" un bit alla variabile *signed* che quindi potrà assumere un numero minore di valori rispetto alla stessa variabile *unsigned*.

Questo programma mostra come la dimensione e i valori minimi e massimi del tipo di dato `int` possano variare con l'utilizzo dei modificatori:

```
/**
 * @file tipi-di-dato-limiti.cpp
 * Mostra i valori possibili per i principali tipi di dato del C++.
 */

#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

/**
 * Funzione template che mostra la dimensione e i
 * valori minimi e massimi per un tipo di dato.
 */
```

```
template<typename T>
void dimensione()
{
    cout << sizeof(T) << " bytes,"
         << setw(4)   << "da:"
         << setw(21)  << numeric_limits<T>::min()
         << setw(4)   << "a:"
         << setw(21)  << numeric_limits<T>::max()
         << endl;
}

int main()
{
    /** Variazione della dimensione e dei valori del tipo int */
    cout << setw(16) << "int: ";
    dimensione<int>();
    cout << setw(16) << "unsigned int: ";
    dimensione<unsigned int>();
    cout << setw(16) << "short int: ";
    dimensione<short int>();
    cout << setw(16) << "unsigned short: ";
    dimensione<unsigned short>();
    cout << setw(16) << "long int: ";
    dimensione<long int>();
    cout << setw(16) << "unsigned long: ";
    dimensione<unsigned long>();

    return 0;
}
```

Se esamini con attenzione il codice scoprirai un'altra caratteristica dei modificatori del C++: se sono riferiti a un dato di tipo intero, è possibile accorciare la dichiarazione omettendo la parola-chiave `int`.
L'output di questo programma, sul mio computer è:

```
% g++ tipi-di-dato-limiti.cpp -o ../out/esempio
% ../out/esempio
           int: 4 bytes, da:          -2147483648  a:           2147483647
  unsigned int: 4 bytes, da:                    0  a:           4294967295
     short int: 2 bytes, da:               -32768  a:                32767
unsigned short: 2 bytes, da:                    0  a:                65535
      long int: 8 bytes, da: -9223372036854775808  a:  9223372036854775807
 unsigned long: 8 bytes, da:                    0  a: 18446744073709551615
```

L'ultima cosa di cui ti devo parlare, a proposito dei tipi di dato del C++ sono i cosiddetti *attributi intermediari*, ovvero: le *funzioni*, gli *array*, i *puntatori* e le *reference*.

Delle *funzioni* parleremo diffusamente in seguito. Quello che ci interessa, qui e ora, è che ogni funzione ha un suo tipo di dato. La funzione `main`, che è la funzione principale di ogni programma, ha come tipo di dato `int` e infatti si chiude con l'istruzione:

```
return 0;
```

La funzione `nome_colore`, all'interno della classe `Colore` è di tipo `void` e, come puoi vedere, non ha valore di ritorno.
Gli *array* sono degli insiemi di dati omogenei e si dichiarano aggiungendo al nome della variabile delle parentesi quadre. La dimensione dell'array deve essere definita al momento della sua dichiarazione; questo può avvenire o in maniera esplicita, inserendo il numero di elementi fra le parentesi quadre:

```
char elementi[10];
```

o assegnando all'array un valore che ne determinerà la dimensione massima, così come abbiamo visto nel primo esempio:

```
char array[] = "abcdefghilmnopqrstuvz";
```

Ogni elemento dell'array è identificato da un indice che va 0 al numero di elementi dell'array meno uno. Nel caso dell'array qui sopra, la lettera `a` avrà indice 0, la `b` avrà indice 2 e così via, fino alla `z`, che avrà indice 20.

I *puntatori* sono come i pit-bull: alla maggior parte delle persone fanno paura, ma chi li conosce sa che sono cani dolcissimi, se li sai trattare. Quello che devi tenere sempre a mente è che il *C++* è solo un modo particolare di vedere la memoria del computer e nella memoria non ci sono caratteri, non ci sono classi, non ci sono immagini e non ci sono film porno, ma solo una lunga sequenza di 1 e di 0:

```
01000011001001110110100001100001011011100110111001101111100100000
01100110011011110111001001110011011001010010000001110100011101 01
01110100011101000110010100100000011001010010000001100100011101 01
01100101001000000110110001100001001000000111001101111010001100101
01110011011100110110000100100000011001010111010011100000001000 00
00001010010010100110010100100000011001100110100101101111011101 10
01101001011100110110001101100101001000000110010001100101011011 10
01110100011100100110111100100000011000010110110000100000011000 11
01101111011100100110010100100000001001110110111000100000001101001
01101100011011000111010101110011011010010110111101101110011001 01
```

L'unico modo per dare un senso a questa catena di valori binarii è di suddividerli in blocchi e assegnare un tipo di dato a ciascun blocco. Per esempio, se suddividi la sequenza binaria qui sopra in blocchi di otto bit:

```
01000011 00100111 01101000 01100001 01101110 01101110 01101111
00100000 01100110 01101111 01110010 01110011 01100101 00100000
```

```
01110100 01110101 01110100 01110100 01100101 00100000 01100101
00100000 01100100 01110101 01100101 00100000 01101100 01100001
00100000 01110011 01110100 01100101 01110011 01110011 01100001
00100000 01100101 01110100 11100000 00100000 00001010 01001010
01100101 00100000 01100110 01101001 01101111 01110010 01101001
01110011 01100011 01100101 00100000 01100100 01100101 01101110
01110100 01110010 01101111 00100000 01100001 01110010 00100000
01100011 01101111 01110010 01100101 00100000 00100111 01101110
00100000 01101001 01101100 01101100 01110101 01110011 01101001
01101111 01101110 01100101
```

e converti ciascun valore nel corrispondente carattere del set ASCII, scopri che una sequenza apparentemente insensata di 1 e di 0 è in realtà l'inizio di una famosa canzone popolare:

```
01000011 =  67 = C
00100111 =  39 = '
01101000 = 104 = h
01100001 =  97 = a
01101110 = 110 = n
01101110 = 110 = n
01101111 = 111 = o
00100000 =  32 =
```

Per identificare i valori all'interno della sequenza di bit, hai bisogno di due informazioni: il tipo di dato che stai puntando e il suo indirizzo in memoria. I puntatori ti permettono di ottenere queste informazioni: il loro valore definisce l'indirizzo di memoria da cui leggere; il loro tipo definisce la dimensione del valore puntato.

```
unsigned char * ptr = valori;
```

Molti pensano ai puntatori come a delle bandierine che ti permettono di identificare un punto specifico della memoria, ma questo è vero solo per i puntatori `void`, che non hanno un tipo di dato associato. Gli altri puntatori, più che una bandierina, sono una sorta di maschera che può scorrere sulla sequenza di bit, isolando ed evidenziando i singoli valori che la compongono:

```
01000011 00100111 01101000 01100001 01101110 01101110 01101111
00100000 011001100110111101110010 01110011 01100101 00100000
01110100 01110101 01110100 01110100 01100101 00100000 01100101
00100000 01100100 01110101
```

Questo programma mostra come la dimensione di un puntatore modifichi il risultato della lettura dei dati:

```
#include <iostream>
#include <bitset>
#include <iomanip>
```

```
using namespace std;

/**
 * Definisce due tipi di dato: uno lungo un byte
 * e uno lungo due byte
 */
typedef unsigned char byte;
typedef short int duebyte;

int main()
{
    /** Crea un array di 80 valori binarii */
    byte valori[] = {
          0b01000011, 0b00100111, 0b01101000, 0b01100001
        , 0b01101110, 0b01101110, 0b01101111, 0b00100000
        , 0b01100110, 0b01101111, 0b01110010, 0b01110011
        , 0b01100101, 0b00100000, 0b01110100, 0b01110101
        , 0b01110100, 0b01110100, 0b01100101, 0b00100000
        , 0b01100101, 0b00100000, 0b01100100, 0b01110101
        , 0b01100101, 0b00100000, 0b01101100, 0b01100001
        , 0b00100000, 0b01110011, 0b01110100, 0b01100101
        , 0b01110011, 0b01110011, 0b01100001, 0b00100000
        , 0b01100101, 0b01110100, 0b11100000, 0b00100000
        , 0b00001010, 0b01001010, 0b01100101, 0b00100000
        , 0b01100110, 0b01101001, 0b01101111, 0b01110010
        , 0b01101001, 0b01110011, 0b01100011, 0b01100101
        , 0b00100000, 0b01100100, 0b01100101, 0b01101110
        , 0b01110100, 0b01110010, 0b01101111, 0b00100000
        , 0b01100001, 0b01110010, 0b00100000, 0b01100011
        , 0b01101111, 0b01110010, 0b01100101, 0b00100000
        , 0b00100111, 0b01101110, 0b00100000, 0b01101001
        , 0b01101100, 0b01101100, 0b01110101, 0b01110011
        , 0b01101001, 0b01101111, 0b01101110, 0b01100101
    };

    /**
     * Crea un puntatore con una dimensione di otto bit e gli
     * assegna come valore l'indirizzo dell'inizio dell'array.
     */
    byte * p1 = valori;

    /**
     * Sposta il puntatore per tutta la lunghezza dell'array,
     * mostrando per ciascun byte il suo valore binario, il
     * suo valore convertito in decimale e la lettera del
     * set ASCII corrispondente a quel valore.
```

```
     */
    for(int i = 0; i < sizeof(valori); i++) {
        cout << setw(4)  << i
             << setw(18) << bitset<8>(*p1)
             << setw(6)  << (int)*p1
             << setw(4)  << (char) *p1 << endl;

        /** Sposta il puntatore al byte successivo. */
        p1++;
    }

    cout << endl;

    /**
     * Crea un nuovo puntatore, stavolta con una
     * dimensione di sedici bit.
     */
    duebyte * p2 = (duebyte*)valori;

    /**
     * Legge di nuovo i valori dell'array,
     */
    for(int i = 0; i < sizeof(valori) / 2; i++) {
        cout << setw(4)  << i
             << setw(18) << bitset<16>(*p2)
             << setw(6)  << (short)*p2
             << setw(4)  << (char) *p2 << endl;

        /** Sposta il puntatore di due byte. */
        p2++;
    }

    return 0;
}
```

Questo è l'output del programma; voglio proprio vedere come farai a farlo entrare nel tuo libro:

```
 0          01000011    67   C
 1          00100111    39   '
 2          01101000   104   h
 3          01100001    97   a
 4          01101110   110   n
 5          01101110   110   n
 6          01101111   111   o
 7          00100000    32
 8          01100110   102   f
```

9	01101111	111	o
10	01110010	114	r
11	01110011	115	s
12	01100101	101	e
13	00100000	32	
14	01110100	116	t
15	01110101	117	u
16	01110100	116	t
17	01110100	116	t
18	01100101	101	e
19	00100000	32	
20	01100101	101	e
21	00100000	32	
22	01100100	100	d
23	01110101	117	u
24	01100101	101	e
25	00100000	32	
26	01101100	108	l
27	01100001	97	a
28	00100000	32	
29	01110011	115	s
30	01110100	116	t
31	01100101	101	e
32	01110011	115	s
33	01110011	115	s
34	01100001	97	a
35	00100000	32	
36	01100101	101	e
37	01110100	116	t
38	11100000	224	?
39	00100000	32	
40	00001010	10	
41	01001010	74	J
42	01100101	101	e
43	00100000	32	
44	01100110	102	f
45	01101001	105	i
46	01101111	111	o
47	01110010	114	r
48	01101001	105	i
49	01110011	115	s
50	01100011	99	c
51	01100101	101	e
52	00100000	32	
53	01100100	100	d
54	01100101	101	e

```
55          01101110   110   n
56          01110100   116   t
57          01110010   114   r
58          01101111   111   o
59          00100000    32
60          01100001    97   a
61          01110010   114   r
62          00100000    32
63          01100011    99   c
64          01101111   111   o
65          01110010   114   r
66          01100101   101   e
67          00100000    32
68          00100111    39   '
69          01101110   110   n
70          00100000    32
71          01101001   105   i
72          01101100   108   l
73          01101100   108   l
74          01110101   117   u
75          01110011   115   s
76          01101001   105   i
77          01101111   111   o
78          01101110   110   n
79          01100101   101   e

 0  0010011101000011 10051   C
 1  0110000101101000 24936   h
 2  0110111001101110 28270   n
 3  0010000001101111  8303   o
 4  0110111101100110 28518   f
 5  0111001101110010 29554   r
 6  0010000001100101  8293   e
 7  0111010101110100 30068   t
 8  0111010001110100 29812   t
 9  0010000001100101  8293   e
10  0010000001100101  8293   e
11  0111010101100100 30052   d
12  0010000001100101  8293   e
13  0110000101101100 24940   l
14  0111001100100000 29472
15  0110010101110100 25972   t
16  0111001101110011 29555   s
17  0010000001100001  8289   a
18  0111010001100101 29797   e
19  0010000011100000  8416   ?
```

```
20  0100101000001010 18954
21  0010000001100101  8293   e
22  0110100101100110 26982   f
23  0111001001101111 29295   o
24  0111001101101001 29545   i
25  0110010101100011 25955   c
26  0110010000100000 25632
27  0110111001100101 28261   e
28  0111001001110100 29300   t
29  0010000001101111  8303   o
30  0111001001100001 29281   a
31  0110001100100000 25376
32  0111001001101111 29295   o
33  0010000001100101  8293   e
34  0110111000100111 28199   '
35  0110100100100000 26912
36  0110110001101100 27756   l
37  0111001101110101 29557   u
38  0110111101101001 28521   i
39  0110010101101110 25966   n
```

Come vedi, i bit sono gli stessi, ma cambiando il tipo di dato associato al puntatore e quindi la sua dimensione, il valore dei dati diventa un altro e la conversione dei numeri in lettere perde di significato.
Le istruzioni: `p1++` e `p2++` incrementano di un'unità il puntatore, ovvero lo spostano in avanti nella memoria del numero di byte corrispondenti alla dimensione del tipo di dato a cui è associato. Il puntatore `p1` è un `unsigned char`, quindi si sposta di otto bit; il puntatore `p12` è uno `short int`, quindi si sposta di sedici bit. È questo il motivo per cui il secondo ciclo di lettura dura la metà del primo: perché la quantità di bit letta per ogni iterazione è doppia. Se mi fossi distratto e avessi copiato il codice del primo ciclo di lettura così com'è, senza dividere per due il numero di cicli, il puntatore avrebbe continuato a "camminare" in avanti nella memoria, oltre i confini dell'array e questo, come vedremo in seguito, non è assolutamente bene.
Le *reference* sono degli alias delle variabili a cui sono associati. Come per i gibboni o i cigni, il legame fra una *reference* e la sua variabile dura tutta la vita. Mentre i puntatori possono avere un valore `NULL` e possono essere associati a variabili differenti nel corso dell'elaborazione, le *reference* devono sempre essere associate a una variabile e quell'associazione non può essere modificata successivamente.

```
/**
 * @file tipi-di-dato-reference.cpp
 * Utilizzo delle reference.
 */

#include <iostream>
```

```
#include <iomanip>

using namespace std;

int main()
{
    /** Definisce una variabile di tipo int */
    int  v = 10;

    /** Definisce una reference alla variabile v */
    int& r = v;

    /** I valori di v e r sono identici perché r è un'alias di v */
    cout << "valori iniziali: v=" << v << ", r=" << r << endl;

    /** Modifica il valore della variabile r */
    r = 20;

    /** Il valore di v sarà modificato di conseguenza */
    cout << "valori finali:   v=" << v << ", r=" << r << endl;

    return 0;
}
```

Compilando ed eseguendo il codice, otterrai:

```
% g++ src/cpp/tipi-di-dato-reference.cpp -o src/out/esempio
% src/out/esempio
valori iniziali: v=10, r=10
valori finali:   v=20, r=20
```

Approfondiremo i pro e i contro delle *reference* quando parleremo delle funzioni.

Dato che ciascun tipo di dato non è altro che un modo per vedere la memoria, è possibile fare delle conversioni da un tipo all'altro. Le conversioni possono essere *implicite* o *esplicite*. Le conversioni *implicite* sono quelle che avvengono quando un valore è copiato fra variabili di tipo compatibile:

```
int  i = 0;
char c = i;
```

In questo caso, assegnamo a una variabile di tipo `char` il valore di una variabile di tipo `int` e il compilatore ce lo lascia fare perché il valore dell'intero può essere salvato senza problemi anche nel singolo byte del `char`. Se volessimo rendere esplicita questa conversione, dovremmo utilizzare la sintassi:

```
int  i = 0;
char c = (char) i;
```

```
char c = char (i);
```

Le due forme sono equivalenti. La prima è quella che il C++ ha ereditato dal C; la seconda è chiamata *notazione funzionale*. Nell'esempio precedente, la variabile `valori` è un puntatore a `unsigned char`, ma abbiamo detto al sistema di considerarla un puntatore a `short int` con l'istruzione:

```
duebyte * p2 = (duebyte*)valori;
```

Il C++ ha anche altri modi per convertire un tipo di dato in un altro, ma siccome si applicano principalmente alle classi, li vedremo in seguito.

Quello che ti ho appena detto del C++ vale anche per il C'hi++. Così come la "realtà" all'interno di un computer altro non è che una sequenza di valori binari che, aggregati fra di loro, costituiscono le entità gestite dal sistema, la realtà in cui noi viviamo altro non è che una manifestazione dell'azione dell'energia dell'Universo sulle particelle che costituiscono tutto ciò che esiste, ovvero gli *spazioni*.
Puoi pensare all'Universo come a una versione tridimensionale di uno schermo di PC. Le immagini che vedi su questo schermo sono costituite da minuscoli puntini colorati, detti *pixel*. Quando lo schermo è spento, i pixel sono neri e non mostrano alcuna immagine; quando lo accendi, l'energia del catodo colpisce i pixel e li rende visibili. Tutto ciò che vedi, all'interno dello schermo, però, è illusorio. O meglio: esiste, ma non è ciò che sembra. Gli elementi delle finestre non sono tridimensionali, anche se hanno delle ombre e quando vedi il puntatore del mouse spostarsi da una finestra all'altra in realtà stai vedendo i pixel dello schermo che cambiano di colore È un'illusione auto-indotta e volontaria: il tuo cervello vede qualcosa, ma finge che sia altro, perché è più comodo così.
La stessa cosa avviene con gli spazioni, solo che invece di essere disposti su una superficie piana, come i pixel dello schermo, gli spazioni sono una matrice tridimensionale e hanno quella che il Maestro Canaro definiva: "esistenza potenziale"; se sono irradiati di energia, acquisiscono massa e quindi esistenza, altrimenti non esistono. Quando il *Big Bang* irradia la sua Energia nell'Universo, questa colpisce gli spazioni dando loro una massa e generando tutto ciò che esiste, dalle nebulose di *Wolf–Rayet* alla tua maestra delle Elementari.
Così come un `char`, `double` o `class` sono solo dei nomi convenzionali per degli insiemi di bit, anche `elio`, `tungsteno` o `cocomero` sono nomi convenzionali per delle quantità di energia. *Energia*, bada bene, non *spazioni*, perché gli spazioni sono fissi; quella che si muove è l'Energia. Quando tu sposti il braccio come hai fatto adesso, non stai realmente spostando il braccio, ma trasferendo l'energia che costituiva il tuo braccio da un gruppo di spazioni a un altro. E anche se restiamo immobili, qui, in questa stanza, l'energia di cui siamo costituiti continua a trasferirsi nella matrice degli spazioni perché il Pianeta in cui viviamo ruota intorno al suo asse e intorno al Sole; il Sole ruota intorno alla Via Lattea e la Via Lattea si muove a sua volta rispetto alle altre galassie. Né io né te siamo, adesso, costituiti dagli stessi spazioni che eravamo pochi secondi fa. Siamo

un'immagine in movimento, come il puntatore del mouse, o come le luci delle decorazioni natalizie. Come disse Sant'Agostino, siamo, allo stesso tempo, reali e illusorii:

> E considerai tutte le cose che sono al di sotto di te e vidi che non si può dire in modo assoluto né che esistono né che non esistono: a loro modo esistono, perché derivano da te, non esistono perché non sono ciò che sei tu: ed esiste veramente ciò che permane immutabile.

Per i *Vedanta*, il Mondo ha la natura dell'Arte perché la realtà dell'arte-fatto è differente dalla realtà dell'arte-fice. Questa similitudine, oltre che per Platone, è valida anche per i linguaggi di programmazione: un oggetto all'interno di un programma esiste, ma non è realmente ciò che rappresenta, anche se a noi fa comodo ritenerlo tale. Come scrisse Ezra Pound, solo i sogni esistono realmente, perché la loro natura irreale rimane tale anche in una realtà illusoria.

Gli spazioni definiscono anche i limiti delle nostre grandezze e delle nostre unità di misura. Pensa a una retta: teoricamente, è un insieme infinito di punti che si estende in due direzioni. I punti, però, sono privi di dimensione, quindi, per fare anche solo un piccolo tratto di retta, ne serve un numero infinito. In sostanza, secondo i matematici, in ciascuna retta coesistono due forme di infinito: quello sincategorematico della lunghezza e quello categoriematico della sequenza di punti. (O forse il contrario: non mi ricordo mai qual'è l'infinito in atto e quale quello potenziale.) Capisci bene che questa è una panzana: l'idea che se moltiplichi all'infinito il nulla tu possa ottenere qualcosa è solo un tentativo dei matematici di giustificare la loro vita sessuale.
La retta è la rappresentazione ideale di un insieme finito di spazioni contigui. È molto, molto sottile, ma una dimensione ce l'ha. Anche il piano, con buona pace di René Guenon, è costituito da un insieme finito di spazioni contigui e per un singolo punto non passano infinite rette, ma un numero finito, per quanto alto, perché la granularità degli spazioni limita il numero delle possibili angolazioni. Stesso discorso vale per i numeri naturali: non li si può estendere all'infinito, ma solo fino al numero totale di spazioni attivi nell'Universo, perché qualsiasi numero superiore indicherebbe una quantità che non esiste e sarebbe quindi insensato.
Gli spazioni definiscono anche l'unità minima di tempo, quella che il Maestro Canaro definiva scherzosamente lo "spazione-tempo", ovvero l'intervallo di tempo necessario a uno spazione per acquisire tutta l'energia di uno degli spazioni con cui è a contatto. È questo il motivo per cui la velocità luce non può superare i trecento milioni di metri al secondo: perché è limitata dal tempo di trasferimento dell'energia fra gli spazioni.

Anche il C'hi++ ritiene simili il Mondo e l'Arte, ma non solo per la loro natura. Il Mondo e l'Arte sono simili perché il loro scòpo è lo stesso.

Struttura dei programmi C++

> Wer den Computer will verstehen,
> Muss in Computers Lande gehen

La libertà sintattica e la concisione dei costrutti sono la caratteristica principale del C++.

Il C++, come tutti i linguaggi, ha una sua sintassi che definisce gli elementi del codice e il loro utilizzo all'interno dei programmi. Negli esempii che abbiamo visto finora, ho utilizzato alcuni di questi elementi senza spiegarti precisamente quale sia il loro ruolo, perché volevo darti un quadro d'insieme delle caratteristiche del linguaggio. Un po' come quando arrivi a una festa e ti presentano gli altri invitati uno dietro l'altro e alla fine l'unica cosa che ti ricordi è il decolleté delle signore. Adesso però, dobbiamo fare un passo indietro ed esaminare questi elementi a uno a uno, cominciando da quello che è l'elemento principale di ciascun programma C++, ovvero la funzione `main`.

```
int main(int argc, char** argv)
{
    return 0;
}
```

Quello qui sopra è il più piccolo programma in C++ che tu possa scrivere. È anche il più inutile, però, perché non fa nulla.
L'esempio qui sotto è altrettanto inutile, ma un po' più complesso:

```
/**
 * @file src/struttura-hello-world.cpp
 * Esempio minimo di programma C++.
 */

#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Il suo output, come forse avrai intuito, è:

```
% g++ struttura-hello-world.cpp -o ../out/esempio
% ../out/esempio
Hello, World!
```

La prima linea di codice:

```
#include <iostream>
```

non è un'istruzione, ma una direttiva per il preprocessore. Il preprocessore è un programma che elabora il codice e lo prepara per la compilazione, ne parleremo in seguito. Per il momento, ti basta sapere che questa istruzione fa sì che nel codice venga incluso il file `iostream.h`, che contiene, fra le altre cose, la dichiarazione dello *stream* `cout`, utilizzato nell'unica istruzione del programma. La seconda linea di codice:

```
int main(int argc, char** argv)
```

definisce la funzione `main`, specificando che avrà come valore di ritorno un `int` e due parametri: un intero e un puntatore a puntatori a `char`. Questi valori servono a gestire i parametri passati da riga di comando: l'intero `argc` (crasi di: *argument count*), specifica il numero di parametri presenti nella chiamata, nome del programma compreso, mentre il parametro `argv` (*argument vector*) è un array di puntatori a tutte le stringhe presenti nella linea di comando.
Il corpo della funzione è racchiuso in una coppia di parentesi graffe `{}`. L'istruzione:

```
std::cout << "Hello, World!" << std::endl;
```

scrive la stringa `Hello World!` sullo *stream* `std::cout` (*standard character output*), che solitamente corrisponde allo schermo del computer. L'operatore `<<`, in questo caso, notifica al sistema di inviare in output quanto si trova alla sua destra e torna una *reference* allo stream di output, in modo da poter essere ripetuto su una stessa linea di codice, che è una forma più efficiente ed elegante di:

```
std::cout << "Hello, World!" ;
std::cout << std::endl;
```

Infine, l'istruzione:

```
return 0;
```

torna il valore `0` al programma chiamante (di solito, la *shell* del computer) per indicare l'assenza di errori nell'elaborazione. Il programma seguente mette in pratica i concetti visti finora:

```
#include <iostream>

#define NO_ERRORI     0
#define NO_PARAMETRI  1

int main(int argc, char** argv)
{
    int errore = NO_ERRORI;
```

```
    /** Se c'è solo il nome del programma, errore */
    if(argc < 2) {
        std::cerr << "Specificare un parametro!" << std::endl;
        return NO_PARAMETRI;
    }

    /** Stampa tutti i parametri ricevuti */
    for(int p=0; p < argc; p++) {
        std::cout << *argv++ << std::endl;
    }

    return 0;
}
```

Come ti ho detto, il C++ è un linguaggio *compilato*, quindi il codice, per poter essere eseguito, deve essere elaborato dal complilatore con il comando della *shell*:

```
% g++ struttura-argc-argv.cpp -o src/out/esempio
```

Il parametro `-o` permette di specificare il nome del file di output, in questo caso: `src/out/esempio`. Se non si definisce questo valore, il compilatore genera un file di nome `a.out`.
Il comando *shell*:

```
% src/out/esempio; echo $?
```

è composto di due istruzioni, separate dal carattere `;`. La prima istruzione esegue il file compilato; la seconda stampa a video il suo valore di ritorno. Se eseguiamo il programma senza parametri, otteniamo un messaggio e il codice di errore `1`:

```
Specificare un parametro!
1
```

Se invece eseguiamo il programma passandogli dei parametri, otteniamo questo:

```
% src/out/esempio un due tre; echo $?
src/out/esempio
un
due
tre
0
```

Nessun programma degno di questo nome ha solo la funzione `main`, ma suddivide il suo lavoro in una serie di funzioni che svolgono compiti precisi e ben definiti. In un programma ben scritto, le funzioni presentano due caratteristiche, che gli anglosassoni e gli anglofili definiscono: *low coupling* e *high cohesion*.
Con il termine *accoppiamento* di due funzioni si intende la quantità di informazioni che la funzione A deve avere riguardo la funzione B per poterla utilizzare.

Ciascuna funzione si aspetta di ricevere una serie ben definita di parametri: la funzione `raddoppiaStipendio`, che abbiamo visto prima, si aspetta di ricevere un solo parametro, di tipo `long`:

```
long raddoppiaStipendio(long stipendio);
```

mentre un'ipotetica funzione `scorporaIVA` potrebbe richiederne due; l'importo dello stipendio e l'aliquota IVA:

```
float scorporaIVA(long stipendio, float aliquota);
```

In entrambi questi casi, tutto ciò di cui ha bisogno una terza funzione per richiamare `raddoppiaStipendio` o `scorporaIVA` è la la loro *interfaccia*, ovvero il numero, il tipo e l'ordine dei parametri da passare. Ora immagina che un programmatore maldestro abbia scritto la funzione `facciQualcosa` che può compiere più azioni distinte, in base ai parametri ricevuti:

```
float facciQualcosa(long stipendio, int azione, float aliquota = 0)
{
    float valore = 0;
    /** Differenzia l'azione in base al valore di azione */
    if(azione == 1) {
        valore = stipendio * 2;
    } else if(azione == 2) {
        valore = stipendio / ((100 + aliquota) / 100);
    }
    return valore;
}
```

Per poter utilizzare questa funzione, non solo dobbiamo conoscere la sua interfaccia, ma dobbiamo anche sapere quali azioni corrispondono ai diversi valori del parametro `azione`. Questa è follia, *meshuggah*, perché, se un giorno l'autore la modificasse e decidesse che il valore `1` del parametro `azione` causa lo scorporo dell'IVA mentre il valore `2` causa il raddoppio dello stipendio, noi dovremmo modificare anche *tutte* le funzioni che l'hanno chiamata per adattarle alle nuove regole. Non solo perderemmo del tempo, ma se dimenticassimo di aggiornare una o più chiamate otterremmo un programma con un funzionamento errato.
Il *coupling* è come il colesterolo: più è basso, meglio è; quindi, per evitare errori, dobbiamo ridurlo, creando un `enum` a cui assegnare i possibili valori del parametro `azione`:

```
enum Azione { raddoppia, scorpora };
```

Come spesso avviene, una singola riga di codice ben scritto ci permette di risparmiare tempo e di ottenere del codice più robusto, perché l'effetto del parametro `azione`, in questo modo, sarà del tutto indipendente dal suo valore numerico:

```
enum Azione { raddoppia, scorpora };

float facciQualcosa(long stipendio, Azione azione, float aliquota = 0)
```

```
{
    float valore = 0;
    /** Agisce in base all'etichetta, non al valore */
    if(azione == raddoppia) {
        valore = stipendio * 2;
    } else if(azione == scorpora) {
        valore = stipendio / ((100 + aliquota) / 100);
    }
    return valore;
}
```

La funzione `facciQualcosa` ha anche un altro difetto progettuale, oltre all'alto accoppiamento: manca di coesione interna. In un programma ben scritto, ciascuna funzione deve avere solo una.. funzione:

```
inline long raddoppiaStipendio(long stipendio)
{
    return stipendio * 2;
}

inline float scorporaIVA(long stipendio, float aliquota)
{
    return (stipendio / ((100 + aliquota) / 100);
}
```

Anche in un esempio così semplice, vedi bene che differenza ci sia, fra una funzione che può svolgere più azioni eterogenee e una funzione che svolge una singola azione, ben precisa. Riducendo la complessità della funzione, inoltre, abbiamo la possibilità di dichiararla come inline, aumentando la velocità di esecuzione del programma.

La funzione `main` del C'hi++ è quello che gli scienziati chiamano: *Big Bang*. Così come l'atmosfera è agitata dallo scontro di masse di aria calda e fredda, l'Energia dell'Universo è costantemente sottoposta all'azione contrapposta di due forze: *Gravità*, che tende ad accorpare tutta la materia nell'Uno primgenio ed *Entropia*, che al contrario, tende a dividere. Poe lo aveva capito. In: *Eureka*, usa il termine *Elettricità* invece di: *Entropia*, ma la contrapposizione che descrive è corretta.
Ciò che non è corretto (e non solo in Poe) è l'idea che l'espansione dell'Universo sia un evento unico. La vita dell'Universo è ciclica: quando è preponderante l'Entropia, l'Universo si espande, come in questo momento; quando "vince" la Gravità, l'Universo collassa su sé stesso e torna all'Uno. Ciò che evita la stasi in uno dei due punti estremi del ciclo — l'Uno e la morte termica — è una terza forza, che il Maestro Canaro chiamava: *l'annosa dicotomia fra ciò che desideriamo è ciò di cui abbiamo bisogno.*
Così come una scatola di mattoncini Lego contiene tutti gli edifici che hai costrui-

to e che costruirai in futuro, nell'Uno primigenio è concentrata tutta l'Energia dell'Universo e quindi ogni essere animato o inanimato che sia mai esistito o che mai esisterà. La Gravità è al suo punto estremo e una non-esistenza scorre in un non-tempo, che non può essere misurato perché non esistono eventi in base a cui farlo. Non ci sono né morte, né sofferenza, né malattia, né separazione; l'Uno è, di fatto, ciò che la maggior parte degli esseri senzienti descrive e auspica come un Universo perfetto.
L'Energia, inizialmente, è soddisfatta, ma a poco a poco l'appagamento per la raggiunta Unità scema e cresce invece il desiderio di qualcosa di diverso da quella cristallina perfezione. Così come di un vecchio amore si ricorda solo ciò che ci fa piacere ricordare, obliandone i difetti, l'Energia ripensa a quando l'Universo non era buio e vuoto, ma risplendeva della luce di innumerevoli stelle e si chiede se, in fondo, non sia quella, l'esistenza a cui ambisce, se non sia quello, in effetti, il Paradiso.
Ha una chiara memoria degli errori e dei dolori delle passate esistenze (è stata lei, incarnata negli spazioni ad averli commessi), ma, come uno scacchista che debba giocare nuovamente una determinata apertura, pensa che stavolta andrà meglio, che non ripeterà gli sbagli fatti in precedenza e questo desiderio di un riscatto genera il *Big Bang*. L'Energia disintegra l'enorme buco nero in cui si era rannicchiata e si espande di nuovo nell'Universo, dando massa agli spazioni e generando la materia. L'Era della Gravità finisce e comincia una nuova Era dell'Entropia; qualcosa di molto simile a quello che trovi descritto nella *Bhagavad-Gita*:

> I cicli cosmici sono periodi temporali chiamati Manvantara, suddivisi al proprio interno in quattro ere o Yuga, ciascuna caratterizzata da una particolare qualità dell'esistenza. Si tratta di un ritorno periodico a condizioni di vita non uguali ma analoghe, da un punto di vista qualitativo, a quelle dei cicli precedenti, una successione di quattro ere che ricorda, su scala ridotta, l'alternarsi delle quattro stagioni.

Anche la Genesi biblica può essere considerata un'allegoria della cosmogonia spazionista (o viceversa): il Paradiso è l'Uno primigenio, mentre Adamo (*Puruṣa*) ed Eva (*Prakṛti*) sono ciò che ne causa la disgregazione, generando un Universo imperfetto e doloroso, aiutati dal , ovvero da colui che *separa* o *scaglia attraverso*, altrimenti noto come: *Luci-fero*.
In ottica spazionista, la domanda:

> Perché, se Dio è buono, nell'Universo che ha creato esistono il male e il dolore?

non ha senso, perché non è Dio ad aver creato l'Universo, ma il Diavolo, così come sostenevano i Barbelognostici.

Una volta, un discepolo chiese al Maestro Canaro come fosse possibile che l'Energia dell'universo avesse dei sentimenti o delle aspirazioni e se questa auto-

coscienza non contrastava con l'idea che il C'hi++ sia una metafisica priva di elementi metafisici. Il Maestro Canaro rispose che il discepolo aveva ragione (un modo ellittico per dire che era un idiota). A lui piaceva credere che fosse così perché era un vecchio sentimentale, ma il discepolo era libero di pensare che fosse solo un artificio retorico, sfruttato per rendere più coinvolgente la narrazione. Spiegò poi che l'unica cosa in cui era necessario credere, anche in assenza di prove, è che l'Universo, alla fine di questa fase di espansione, sarebbe tornato a collassare nell'Uno.
A quel punto, i casi sarebbero stati due: o sarebbe rimasto Uno per il resto del Tempo (ipotesi lecita, ma noiosa) oppure sarebbe esploso di nuovo, dando vita a un nuovo Universo. Anche in questo caso le ipotesi sarebbero state due. La prima è che un Big Bang possa avvenire solo in determinate condizioni e che quelle condizioni portino necessariamente a un Universo identico a quello come noi lo conosciamo adesso; quindi, se l'Uno esploderà di nuovo, ricomincerà tutto da capo. La seconda ipotesi è che ogni *Big Bang* avviene in circostanze e con modalità specifiche e che quindi, se l'Uno esploderà di nuovo, nascerà un nuovo Universo, che potrà avere pochi o nessun punto di contatto con quello corrente.
Il Maestro Canaro disse che la prima ipotesi era possibile, ma poco probabile e che quindi avremmo dato per scontato che fosse la seconda, quella corretta: «Tanto, non cambia niente: se il tempo che abbiamo a disposizione per far esplodere e implodere l'Universo è infinito, per quanto bassa possa essere la probabilità che si verifichino due esplosioni uguali è impossibile che la cosa o prima o poi non avvenga. Come dice quel senza-Dio di Dawkins:

> Dato un tempo infinito o un numero di opportunità infinite, è possibile qualsiasi cosa.

In base allo stesso principio, dando per scontato che o prima o poi questo Universo tornerà a manifestarsi, è del tutto lecito pensare — non per fede, ma in base a un banale calcolo probabilistico — che anche ciò che c'è in esso possa o prima o poi tornare a essere. Noi compresi».

Gli operatori

```
question = 2b | ( ! 2b );
```

La varietà e soprattutto l'adattabilità degli operatori sono la caratteristica principale del C++.

Gli operatori sono degli elementi del linguaggio che permettono, appunto, di *operare* delle azioni sulle variabili o sui valori. Li possiamo classificare o in base al numero di operandi su cui agiscono:

- primarii

- unarii
- binarii
- ternarii

o, in maniera più funzionale al tuo libro, in base al tipo di operazione che compiono:

- aritmetici
- logici
- di relazione
- bitwise
- assegnazione

Gli operatori *aritmetici* sono quelli che permettono di eseguire delle comuni operazioni di addizione, sottrazione, divisione o moltiplicazione sulle variabili:

nome	descrizione	esempio
+	addizione	x + y
-	sottrazione	x - y
*	moltiplicazione	x * y
\	divisione	x / y
%	modulo	x % y
++	incremento	x++
–	decremento	y–

L'operatore modulo `%` permette di calcolare il resto della divisione fra due interi (per esempio: `15 % 4 = 3`) e non può essere utilizzato con numeri in virgola mobile.
Gli operatori di incremento `++` e decremento `--` permettono di aumentare o di diminuire di un'unità il valore di una variabile. Quando questi operatori precedono la variabile, l'incremento o il decremento è calcolato immediatamente; se invece compaiono dopo la variabile, l'operazione di incremento o decremento avviene dopo il suo utilizzo:

```
/**
 * @file _man/src/operatori-pre-post.cpp
 * Utilizzo degli operatori di incremento e decremento.
 */

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    /** Definizione delle due variabili */
```

```
    int pre  = 5, post = 7;

    /** Operatore di incremento, prima e dopo la variabile */
    cout << "pre-incremento: " << ++pre  << endl;
    cout << "post-incremento:" << post++ << endl;

    /** Operatore di decremento, prima e dopo la variabile */
    cout << "pre-decremento: " << --pre  << endl;
    cout << "post-decremento:" << post-- << endl;

    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/operatori-pre-post.cpp -o src/out/esempio
> src/out/esempio
pre-incremento: 6
post-incremento:7
pre-decremento: 5
post-decremento:8
```

Gli operatori *di relazione* permettono di verificare il rapporto fra due variabili:

nome	descrizione	esempio
==	uguaglianza	x == x
!=	differenza	x != y
>	maggiore di	x > y
<	minore di	x < y
>=	maggiore o uguale a	x >= y
<=	minore o uguale a	x <= y

Questi operatori tornano un valore booleano, vero o falso a seconda che la condizione sia verificata o no.
L'operatore `==` torna `true` se gli operandi sono uguali, come nel verso:

> Io vidi il mio Signore con l'occhio del cuore.
> Gli chiesi: Chi sei?
> Rispose: Te!

L'operatore `!=` torna `true` se gli operandi non sono uguali; gli operatori `>` e `<` tornano `true`, rispettivamente, se l'operatore a sinistra è maggiore o minore dell'operando a destra; gli operatori `>=` e `<=` tornano `true` se l'operando a sinistra è maggiore o uguale oppure minore o uguale a quello a destra.
Gli operatori *logici* permettono di compiere delle operazioni di logica booleana sulle variabili:

nome	descrizione	esempio
&&	AND logico	x && y
||	OR logico	x || y
!	NOT logico	!(x && y)

L'operatore `&&` torna `true` se entrambe le variabili sono `true`; l'operatore `||` torna `true` se almeno una delle variabili è `true`; l'operatore `!` inverte il valore dell'operando: se è `true`, torna `false` e viceversa.
Gli operatori binarii, o: *bitwise* permettono di effettuare delle operazioni sui valori binarii delle variabili:

nome	descrizione	esempio
&	AND	x & y
|	OR (inclusivo)	x | y
^	XOR (esclusivo)	x ^ y
~	NOT	x ~ y
»	shift a destra	x » y
«	shift a sinistra	x « y

L'operatore `&` imposta a 1 un bit nel risultato se quel bit è 1 in entrambi gli operandi. L'operatore `|` imposta a 1 un bit nel risultato se quel bit è 1 in uno dei due operandi. L'operatore `^` imposta a 1 un bit nel risultato se quel bit è 1 in uno dei due operandi, ma non nell'altro. L'operatore `~` inverte i bit dell'operando. Gli operatori `<<` e `>>` spostano rispettivamente a sinistra e a destra i bit dell'operando di destra per il numero di bit specificato dall'operando di destra.
Gli operatori *di assegnazione* eseguono le operazioni che abbiamo visto finora e, in più, assegnano il valore risultante all'operando di sinistra:

nome	descrizione	esempio
=	uguaglianza	x = x
+=	somma	x += y
-=	differenza	x -= y
*=	moltiplicazione	x *= y
/=	divisione	x /= y
%=	modulo	x %= y
&=	AND	x &= y
|=	OR inclusivo	x |= y
^=	OR esclusivo	x ^= y
»=	shift a destra	x <= y
«=	shift a sinistra	x <= y

Per esempio, l'esperessione:

```
x *= y
```

equivale a:

```
x = (x * y);
```

Il prossimo esempio mostra l'utilizzo e il risultato di ciascun operatore:

```
/**
 * @file operatori-utilizzo.cpp
 * Esempio di utilizzo degli operatori.
 */

#include <iostream>
#include <iomanip>
#include <bitset>

using namespace std;

int main(int argc, char** argv)
{
    int x = 12;
    int y = 4;

    cout << "Operatori aritmetici" << endl;
    cout << "x + y = " << x + y << endl;
    cout << "x - y = " << x - y << endl;
    cout << "x * y = " << x * y << endl;
    cout << "x / y = " << x / y << endl;
    cout << "x % y = " << x % y << endl;
    cout << "x++   = " << x++ << endl;
    cout << "x--   = " << x-- << endl;
    cout << endl;

    cout << "Operatori logici" << endl;
    cout << "x && y    = " << (x && y)  << endl;
    cout << "x || y    = " << (x || y)  << endl;
    cout << "!(x && y) = " << !(x && y) << endl;
    cout << endl;

    cout << "Operatori di relazione"  << endl;
    cout << "x == y = " << (x == y)  << endl;
    cout << "x != y = " << (x != y)  << endl;
    cout << "x > y  = " << (x  > y)  << endl;
    cout << "x < y  = " << (x  < y)  << endl;
    cout << "x >= y = " << (x >= y)  << endl;
```

```
    cout << "x <= y = " << (x <= y)  << endl;
    cout << endl;

    cout << "Operatori bitwise"  << endl;
    cout << "bit x  = " << bitset<8>(x) << endl;
    cout << "bit y  = " << bitset<8>(y) << endl;
    cout << "x & y  = " << bitset<8>(x & y)  << endl;
    cout << "x | y  = " << bitset<8>(x | y)  << endl;
    cout << "x ^ y  = " << bitset<8>(x ^ y)  << endl;
    cout << "x ~ y  = " << bitset<8>(~ y)    << endl;
    cout << "x >> y = " << bitset<8>(x >> y) << endl;
    cout << "x << y = " << bitset<8>(x << y) << endl;
    cout << endl;

    cout << "Operatori di assegnazione"  << endl;
    cout << "x += y = " << (x += y) << endl;
    cout << "x -= y = " << (x -= y) << endl;
    cout << "x *= y = " << (x *= y) << endl;
    cout << "x /= y = " << (x /= y) << endl;
    cout << "x %= y = " << (x %= y) << endl;
    cout << endl;

    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/operatori-utilizzo.cpp -o src/out/esempio
> src/out/esempio
Operatori aritmetici
x + y = 16
x - y = 8
x * y = 48
x / y = 3
x % y = 0
x++   = 12
x--   = 13

Operatori logici
x && y    = 1
x || y    = 1
!(x && y) = 0

Operatori di relazione
x == y = 0
x != y = 1
x > y  = 1
```

```
x < y  = 0
x >= y = 1
x <= y = 0

Operatori bitwise
bit x  = 00001100
bit y  = 00000100
x & y  = 00000100
x | y  = 00001100
x ^ y  = 00001000
x ~ y  = 11111011
x >> y = 00000000
x << y = 11000000

Operatori di assegnazione
x += y = 16
x -= y = 12
x *= y = 48
x /= y = 12
x %= y = 0
```

Adesso dimmi: quanto valgono, le variabili `x` e `y`, alla fine del programma?

Sono degli operatori anche i simboli: `sizeof`, `,`, `.`, `->`, `&`, `*`, `()` e `()?:`. L'operatore `sizeof` lo abbiamo già visto parlando della dimensione dei tipi di dato, perché torna, appunto, la dimensione, in byte, della variabile o del tipo di dato che riceve come parametro.
L'operatore di *cast* `()` permette di modificare il tipo di una variabile e lo abbiamo già utilizzato nella classe `Animale`, quando abbiamo parlato del polimorfismo:

```
const char getSesso() const {
    return (char)_sesso;
}
```

L'operatore condizionale `()?:` è l'unico operatore ternario del C++ e permette di scegliere fra due espressioni a seconda dell'esito di una condizione. La sua sintassi è:

```
(<condizione>) ? <istruzione vero> : <istruzione falso>
```

Scrivere:

```
x = (a > b) ? 4 : 5;
```

è un modo elegante di scrivere:

```
if(a > b) {
    x = 4;
```

```
} else {
    x = 5;
}
```

L'operatore virgola `,` serve ad alterare il modo in cui vengono valutate le espressioni. Quando due o più espressioni sono separate dall'operatore virgola, i risultati delle espressioni a sinistra sono ignorati e viene mantenuto solo il risultato dell'espressione più a destra. In questa espressione, per esempio, alle variabili `a` e `b` è prima assegnato il valore 1 e poi la variabile `b` è incrementata di 2:

```
a = b = 1, b = b + 2;
```

il valore finale di `a` e `b` sarà quindi di 1 e 3, mentre nell'espressione qui sotto sarà di 1 e 2:

```
a = b = 1, b = b + 2, b = b - 1;
```

Gli operatori `.` e `->` servono a identificare i membri di una classe o di un altro dato aggregato. Se ti ricordi (ne dubito), li abbiamo visti per la prima volta parlando del polimorfismo e li utilizzeremo quando ci occuperemo delle classi. Gli operatori `&` e `*`, infine, tornano, rispettivamente, l'indirizzo di memoria di una variabile e un puntatore a una variabile:

```
/**
 * @file operatori-altri.cpp
 * Altri operatori del C++.
 */

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    int     a = 9;
    int     b = 4;

    /** Converte un valore intero in char */
    char    c = char(a * b);

    /** Assegna a t il valore più alto fra a e b */
    int     t = (a > b) ? a : b;

    /** Salva nel puntatore p l'indirizzo di memoria di a */
    int*    p = &a;

    /** Calcola la dimensione di un intero */
    size_t sa = sizeof(a);
```

```
    /** Calcola la dimensione di un puntatore a intero */
    size_t sp = sizeof(p);

    /** Mostra l'indirizzo della variabile a */
    cout << "&a: " << p  << endl;

    /** Mostra la dimensione delle variabili a e p */
    cout << "sa: " << sa << endl;
    cout << "sp: " << sp << endl;

    /** Mostra il risultato dell'operazione ternaria */
    cout << "t : " << t  << endl;

    /** Il codice ASCII 36 corrisponde alla lettera $x\ */
    cout << "c : " << c  << endl;

    cout << endl;

    return 0;
}
```

Se compili ed esegui questo codice, otterrai:

```
> g++ src/cpp/operatori-altri.cpp -o src/out/esempio
> src/out/esempio
&a: 0x7ffee65bba5c
sa: 4
sp: 8
t : 9
c : $
```

L'ultimo operatore di cui dobbiamo parlare è anche quella con il nome più lungo: l'operatore di *risoluzione del campo d'azione* `::`. Oltre ad altri utilizzi connessi con la gestione dei dati delle classi, di cui parleremo in seguito, questo operatore permette di riferirsi a delle variabili con visibilità globale anche nei casi in cui queste vengano oscurate dalla ridefinizione di variabili locali con lo stesso nome:

```
/**
 * @file _man/src/operatori-risoluzione.cpp
 * Utilizzo dell'operatore di risoluzione.
 */

#include <iostream>

using namespace std;
```

```
const char * Stringa = "Stringa globale" ;

int main(int argc, char** argv)
{
    const char * Stringa = "Stringa locale" ;

    /** Stampa la stringa locale */
    cout << Stringa   << endl ;

    /** Utilizza l'operatore per riferirsi alla variabile globale */
    cout << ::Stringa << endl ;

    return 0;
}
```

Compilando ed eseguendo questo codice, ottieni:

```
> g++ src/cpp/operatori-risoluzione.cpp -o src/out/esempio
> ./src/out/esempio
Stringa locale
Stringa globale
```

La definizione dei concetti di *vero* e di *falso* è sempre frutto di un arbitrio. È inevitabile. Per i filosofi è *vero* tutto ciò che può essere descritto con le parole; per gli scienziati, tutto ciò che può essere descritto dalla matematica; per gli informatici è *vero* tutto quello che può essere descritto dal codice. Possiamo definire *falso* come il contrario di *vero*, ma non possiamo dedurre o inferire il concetto di *vero* perché, quale che sia la definizione prescelta, prima di poterla prendere per buona dovremo poterla considerare vera, ma ciò è impossibile perché non sappiamo ancora cosa sia, effettivamente, *vero*.
Bertrand Russell disse che la condizione di *vero* o di *falso* è uno stato dell'organismo, determinato da condizioni esterne all'organismo; è un'affermazione vera.. o, meglio, *valida* anche per i computer: i valori booleani *vero* e *falso* sono due stati possibili per un'istruzione e spesso dipendono da condizioni esterne all'istruzione stessa, come la presenza di un file o un'azione dell'utente. La cosa interessante è che, anche per i computer, i valori che vengono interpretati come *vero* e *falso* sono del tutto arbitrarii e spesso specifici per un dato sistema. Prendi per esempio i programmi che gestiscono le basi di dati: per *Access*, il valore booleano *false* è `-1`; per le prime versioni di *MySQL* è `f`; per Postgres o Oracle, è `0`.
Il computer non ha *in sé* il concetto di *vero* o di *falso*, ma solo l'associazione dei valori booleani *true* e *false* a una determinata sequenza di bit. *Vero* e di *falso*, quindi, sono dei valori simbolici che noi inseriamo nella logica del computer, traducendo nel suo linguaggio un concetto che è proprio del nostro mondo; un riferimento esogeno che può generare delle apparenti contraddizioni come, per esempio, il fatto che tutti i valori digitali siano basati su grandezze analogiche

(la carica magnetica di un nastro, un'infossatura nella superficie di un CD o la tensione all'interno di un circuito), o dei veri e proprii paradossi, come il fatto che, nelle schede perforate, il valore `1` sia associato al vuoto del foro, mentre il valore `0` sia associato alla presenza della carta; di fatto, una negazione logica della realtà.
Questi paradossi, apparenti o reali che siano, non causano alcun problema al sistema, a patto vengano condivisi da tutti gli attori al suo interno. Il paradosso delle schede perforate (buco = 0, carta = 1) era condiviso sia dalle macchine Hollerith che dalle perforatrici delle schede, perciò, le elaborazioni che ne derivavano erano corrette, indipendentemente dal valore relativo attribuito ai concetti di *vero* e di *falso* all'interno del sistema. Da questo punto di vista, non possiamo che dare ragione a William James, quando dice che un'idea è *vera* fintanto che credere in essa è utile per le nostre vite.

Anche gli esseri umani, come i computer, devono riferire le proprie convinzioni a schemi di valori preesistenti perché buona parte dei principii e dei valori su cui modelliamo la nostra e l'altrui esistenza non hanno un riscontro oggettivo nei fatti: sono solo convinzioni o convenzioni che abbiamo deciso di adottare, così come il fatto che `true` equivalga al valore binario `000000001` e `false` a `00000000`.
Prendi la ricchezza, per esempio: si dice che i soldi non danno la felicità ed è vero; nemmeno essere poveri rende felici, ma questo non lo dice nessuno. Comunque, una delle persone più ricche che ho conosciuto non poteva avere figli. Possedeva case in diversi Paesi, uno yacht e perfino un aereo personale, ma non poteva avere qualcosa che anche l'uomo più povero della Terra può facilmente ottenere. Che senso aveva, la sua ricchezza? Era reale?
Letteralmente, si definisce: *prestigio* qualcosa che sembra ciò che non è, come i giochi degli illusionisti, mentre *successo* non è che il participio passato del verbo *succedere*: tutto ciò che è avvenuto in passato è, per definizione, "successo"; malgrado ciò, molte persone, sotto l'influsso dell'Annosa Dicotomia, dedicano la propria vita alla ricerca o dell'uno o dell'altro o di entrambi. La *fama* è altrettanto aleatoria: quante persone sono state famose e ora sono dimenticate? Gli inventori della ruota e del fuoco hanno cambiato per sempre la storia della nostra specie e del mondo con le loro scoperte, ma nessuno sa chi siano stati. In tempi più recenti, il pittore Giovanni Baglione fu molto famoso, alla fine del sedicesimo Secolo, ma se ci ricordiamo di lui oggi è solo per via dei suoi rapporti con Caravaggio e, comunque, la loro fama terminerà con la nostra specie, insieme a quella di Shakespeare, Leonardo o Einstein.
Non sono inconsistenti solo i valori mondani, come fama successo o denaro, ma anche quelli che consideriamo usualmente "nobili", come il rispetto della vita altrui. Se tu mi chiedi perché non si debba uccidere un uomo, ti risponderò che uccidere è sbagliato perché tutto il male che fai, lo stai facendo a te stesso; penso che sia vero perché me lo ha insegnato il Maestro Canaro, che aveva visto il codice del programma dell'Universo. Se invece lo chiedi a un cristiano o a un ebreo, lui ti risponderà che è vietato dai Dieci Comandamenti che Dio ha

dato a Mosè; loro pensano che sia vero perché è scritto nella Bibbia, che è la Parola di Dio. Se infine lo chiedi a un paladino del laicismo, ti risponderà che l'omicidio, per il nostro ordinamento giuridico, è un reato; ciò dev'essere vero, perché le nostre Leggi applicano i concetti espressi nella nostra Costituzione la quale, a sua volta, si rifà ai principii di libertà e uguaglianza della *Dichiarazione dei Diritti dell'Uomo e del Cittadino* francese, che a sua volta si rifaceva alla *Dichiarazione di Indipendenza* americana, che a sua volta riprendeva le idee di Locke e Montesquieu.
Nessuna di queste affermazioni è valida di per sé: sono tutte fondate su una qualche forma di fede — religiosa o laica che sia — in chi ha promulgato la Legge o il Principio.
È, applicato all'etica, lo stesso meccanismo che si utilizza per certificare un server Web. Quando tu accedi al sito Web della tua banca, sai che è davvero il server della tua banca perché possiede un certificato digitale che attesta la sua identità, firmato digitalmente da un ente certificatore detto *Certification Authority* (o, più brevemente: *CA*). Tu sai che l'ente certificatore è davvero chi dice di essere perché anche lui ha un suo certificato, firmato da un'altro ente certificatore, chiamato: *root CA*. La *root CA* garantisce per la CA intermedia che a sua volta garantisce per la tua banca. Sfortunatamente, però, né i legislatori né i loro certificatori ideologici sono sottoposti alle regole e ai controlli che deve rispettare una *Certification Authority* e spesso è accaduto che chi ha definito delle regole sia stato anche il primo a non tenerne conto.
Mosè, con il sesto Comandamento ancora fresco di stampa, chiese ai figli di Levi:

> Ognuno di voi si metta la spada al fianco; percorrete l'accampamento da una porta all'altra di esso, e ciascuno uccida il fratello, ciascuno l'amico, ciascuno il vicino!

Gli Americani, che scrissero:

> We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness.

commerciavano in schiavi e hanno sterminato i Nativi Americani per rubar loro le terre. I Francesi, il cui motto era:

> Liberté, Égalité, Fraternité

hanno applicato questi principii a colpi di ghigliottina. L'ONU, che nel 1948 ha pubblicato la *Dichiarazione Universale dei Diritti Umani* ha al suo vertice dei Paesi che sono i principali produttori di armi del Mondo e che violano costantemente quelle stesse regole da loro promosse.
Attenzione, però: il fatto che i principii laici di pace, uguaglianza e libertà derivino, in ultima analisi, dalle convinzioni di un gruppo di credenti, non vuol dire che siano sbagliati, ma che non gli si può attribuire nessuna veridicità oggettiva; possiamo solo accettarli per fede, così come i dogmi religiosi. *Scegliamo* di credere in uno schema di valori, così come potremmo scegliere di credere negli

UFO: non abbiamo convinzioni innate, se non l'egoismo.
Le persone per bene hanno una cosa in comune con i malandrini: pensano che il resto del Mondo sia come loro, ma si sbagliano. Noi troviamo condivisibile, quasi pleonastico, che ciascun essere umano abbia diritto alla vita, alla libertà e a una giustizia imparziale, perché siamo stati educati con questi valori, ma qualcuno con un differente livello culturale potrebbe non ritenere queste affermazioni altrettanto evidenti *in sé*.
Per gli autori della *Dichiarazione di Indipendenza* era *self-evident* che tutti gli uomini fossero stati creati uguali, ma sei Stati del Sud, nelle loro Costituzioni, precisarono che si stava parlando di uomini liberi, non degli schiavi.
I nazisti utilizzarono le macchine Hollerith per censire e sterminare gli ebrei; furono anche aggiunte delle colonne alle schede perforate per poter gestire i dati relativi alla religione. Noi oggi critichiamo sia loro sia chi li aiutò a farlo, ma forse i nostri nipoti criticheranno noi per la tolleranza che abbiamo mostrato nei confronti di chi ci vende il petrolio per le nostre automobili; la stessa tolleranza che loro mostreranno a chi li rifornirà di materie prime per le batterie, o di acqua.

Scegliamo un ethos perché rende migliore la nostra vita, non perché ci sta simpatico l'autore. Il Maestro Canaro, per esempio, era convinto che il Libro dell'Esodo fosse il resoconto di un esperimento alieno su una popolazione isolata nel deserto per due generazioni ("capisci: la circoncisione serviva a identificare i maschi e l'appartenenza al gruppo era stabilita in base alla madre perché così era verificabile grazie ai mitocondri"); ciò non ostante, regolava la sua vita e le sue decisioni in base a una sua versione "laicizzata" dei dieci Comandamenti di Mosè:

1. Il senso della vita è capire il senso della vita.
2. Non cercare il senso della vita nelle cose terrene.
3. Ciò che conta è come ti comporti, non quello che dici.
4. Dedica parte del tuo tempo alla ricerca spirituale (v. punto 1).
5. Non giudicare i tuoi genitori finché non ti sarai trovato in situazioni simili alle loro.
6. Non uccidere senza un valido motivo (per definire il concetto di "valido motivo" v. punti 1-10).
7. Ci sono tante donne non sposate: tròmbati quelle.
8. Non prendere ciò che non ti appartiene, di qualunque cosa si tratti.
9. Non fare falsa testimonianza contro il tuo prossimo.
10. Non desiderare ciò che appartiene ad altri; se riesci a non desiderare nulla è anche meglio.

Quando gli chiesi perché avesse scelto proprio quelle regole, mi rispose: "Se li avessi seguiti anche da giovane, adesso avrei molti rimorsi in meno."

Il preprocessore

Tutte le potenze, tranne quella di 1, crescono

Le direttive al preprocessore sono la caratteristica principale del C++.

Negli esempi precedenti abbiamo visto alcune istruzioni particolari, perché hanno un carattere `#` all'inizio e non hanno un carattere `;` alla fine:

```
#include <iostream>

#define NO_ERRORI    0
#define NO_PARAMETRI 1
```

Queste istruzioni sono delle *direttive al pre-processore* e possono essere di tre tipi:

- direttive di inclusione;
- definizioni e macro-istruzioni;
- direttive condizionali.

Le *direttive di inclusione* sono quelle che si utilizzano più comunemente e servono a importare nel codice le definizioni delle funzioni di libreria, delle macro-istruzioni e dei simboli necessari per la corretta compilazione dei programmi. Questi elementi, per comodità, sono isolati all'interno di alcuni file, detti *file di include*. Quando il preprocessore incontra una direttiva `include`, la sostituisce con il contenuto del file a cui fa riferimento. Per esempio, se più di un programma dovesse usare la Classe `Colore` o la struttura `ColoreRGB` che abbiamo utilizzato nel programma che visualizza la dimensione dei principali tipi di dato del C++, questi dovrebbero essere isolati in un file separato con estensione `.h`, a indicare che si tratta di un *header file*:

```
#ifndef _CLASS_COLORE
#define _CLASS_COLORE 1

#define  RGB_RED    0xFF0000
#define  RGB_GREEN  0x00FF00
#define  RGB_BLUE   0x0000FF

#ifdef LANG_IT
    #define  STR_RGB    "colore RGB"
    #define  STR_RED    "rosso"
    #define  STR_GREEN  "verde"
    #define  STR_BLUE   "blu"
#else
    #define  STR_RGB    "RGB color"
```

```
    #define  STR_RED    "red"
    #define  STR_GREEN  "green"
    #define  STR_BLUE   "blue"
#endif

#include <iostream>

using namespace std;

/** Definisce un tipo di dato enumerato di nome RGB */
enum RGB { red = RGB_RED, green = RGB_GREEN, blue = RGB_BLUE };

/** Definisce una struttura che contiene un colore RGB e un nome */
struct ColoreRGB {
    RGB  valore;
    const char* nome;
};

/**
 * Definisce la classe Colore, che contiene una struct
 * coloreRGB e una funzione che ne mostra il nome.
 */
class Colore {
public:
    ColoreRGB coloreRgb;
    void nome_colore() {
        cout << STR_RGB << ": ";
        switch(coloreRgb.valore) {
            case red  : cout << STR_RED  ; break;
            case green: cout << STR_GREEN; break;
            case blue : cout << STR_BLUE ; break;
        }
        cout << endl;
    }
};

#endif /* _CLASS_COLORE */
```

Il file verrebbe poi incluso nel codice dei programmi che ne fanno uso con una direttiva `include`:

```
/**
 * @file src/preprocessore-main.cpp
 * Mostra i valori possibili per i principali tipi di dato del C++.
 */

#include <iostream>
```

```
#include <iomanip>

#define LANG_IT
#include "colore.h"
#undef LANG_IT

#ifdef LANG_IT
    #define  STR_BOOL   "booleano"
    #define  STR_CHAR   "carattere"
    #define  STR_INT    "intero"
    #define  STR_DEC    "decimale"
    #define  STR_ARRAY  "array"
    #define  STR_VERDE  "verde"
#else
    #define  STR_BOOL   "boolean"
    #define  STR_CHAR   "character"
    #define  STR_INT    "integer"
    #define  STR_DEC    "decimal"
    #define  STR_ARRAY  "array"
    #define  STR_VERDE  "green"
#endif

using namespace std;

int main()
{
    /** Dichiara una serie di variabili */
    bool    booleano = false;
    char    carattere = 'C';
    int     intero = 1234567890;
    float   decimale = 3.14;
    char    array[] = "abcdefghilmnopqrstuvz";
    RGB     enumerato = green;

    /** Crea un oggetto di tipo Colore */
    Colore  colore;

    /**
     * Assegna un valore ai dati della struttura coloreRgb
     * all'interno dell'oggetto di tipo Colore.
     */
    colore.coloreRgb.valore = enumerato;
    colore.coloreRgb.nome = STR_VERDE;

    /** Mostra il valore delle variabili */
    cout << setw(10) << STR_BOOL  << ": " << booleano   << endl;
```

```
    cout << setw(10) << STR_CHAR  << ": " << carattere  << endl;
    cout << setw(10) << STR_INT   << ": " << intero     << endl;
    cout << setw(10) << STR_DEC   << ": " << decimale   << endl;
    cout << setw(10) << STR_ARRAY << ": " << array      << endl;

    /** Mostra il nome del colore */
    colore.nome_colore();

    return 0;
}
```

Questo esempio utilizza due forme distinte per la direttiva `include`:

```
#include <iostream>
#include "colore.h"
```

La prima forma serve a includere i file di sistema, come, appunto, `iostream`, che contiene le definizioni degli *stream* standard del C++; la seconda forma si utilizza per i file specifici dell'applicazione; nel nostro caso, `colore.h`.
I file di include possono includere a loro volta altri file; per esempio, `colore.h` include `iostream`, perché utilizza lo *stream* `cout`. Anche il nostro codice di esempio, però, include `iostream` e questo potrebbe causare un errore di compilazione se il precompilatore effettuasse due volte l'inclusione, perché sarebbe come se dichiarassimo due volte la stessa funzione. Per questo motivo, all'inizio del nostro file di include (ma anche di `iostream`) troviamo un altro tipo di direttive al preprocessore, le *direttive condizionali*:

```
#ifndef _CLASS_COLORE
#define _CLASS_COLORE 1

...

#endif /* _CLASS_COLORE */
```

Le direttive condizionali sono:

direttiva	valore
`#if`	se non zero
`#ifdef`	se definito
`#ifndef`	se non definito
`#else`	altrimenti
`#elif`	altrimenti se
`#endif`	fine del blocco condizionale

Quando il preprocessore legge la prima direttiva nel file di include, verifica che sia definito un valore per `_CLASS_COLORE`. Se `_CLASS_COLORE` non ha un valore associato, il preprocessore esegue l'istruzione successiva, che gli assegna il valore

1, poi inserisce nel file chiamante tutto il codice fino all'istruzione `#endif`. Se invece `_CLASS_COLORE` ha già un valore associato perché è già stata inclusa da altri file, il preprocessore salta direttamente alla direttiva `#endif` senza riscrivere le tre dichiarazioni.

Le istruzioni seguenti definiscono delle costanti numeriche per i colori dell'`enum RGB`:

```
#define  RGB_RED    0xFF0000
#define  RGB_GREEN  0x00FF00
#define  RGB_BLUE   0x0000FF
```

Le direttive al preprocessore permettono di definire anche delle costanti stringa:

```
#define  STR_GREEN  "verde"
```

Questa è una buona cosa, perché, come imparerai con l'esperienza, avere delle stringhe *hardcoded* all'interno dei programmi causa sempre dei problemi e soprattutto lega il tuo codice a un determinato linguaggio:

```
colore.coloreRgb.nome = "verde";

/** Mostra il valore delle variabili */
cout << "booleano:"    << booleano   << endl;
cout << "carattere:"   << carattere  << endl;
cout << "intero:"      << intero     << endl;
cout << "decimale:"    << decimale   << endl;
cout << "array:"       << array      << endl;
```

Questo può essere accettabile in un programma di esempio, ma è una scelta miope per un programma reale, specie se le stringhe si ripetono in contesti diversi:

```
char stringa[] = "ebete";

...

cout << 6 << "ebete" << endl;
```

perché se la stringa dovesse variare (e stai pur certo che succederà), tu dovrai modificare tutte le righe di codice in cui compare. Al contrario, se definisci delle costanti per tutte le stringhe che utilizzi nel tuo codice, la correzione sarà unica:

```
#define STR_COME_SEI = "astuto";

char stringa[] = STR_COME_SEI;

...

cout << 6 << STR_COME_SEI << endl;
```

Unite alle direttive condizionali, le definizioni di costanti stringa ti permettono di avere un codice multi-lingua:

```
#ifdef LANG_IT
    #define  STR_RGB    "colore RGB"
    #define  STR_RED    "rosso"
    #define  STR_GREEN  "verde"
    #define  STR_BLUE   "blu"
#else
    #define  STR_RGB    "RGB color"
    #define  STR_RED    "red"
    #define  STR_GREEN  "green"
    #define  STR_BLUE   "blue"
#endif
```

La definizione della costante che determina la condizione (in questo caso, `LANG_IT`) può avvenire o nel codice del programma che include il file con definizioni:

```
#define LANG_IT
#include "colore.h"
```

o direttamente da riga di comando, come parametro di compilazione:

```
% g++ ./cpp/preprocessore-main.cpp -D LANG_IT -o ./out/esempio
% ./out/esempio
  booleano: 0
 carattere: C
    intero: 1234567890
  decimale: 3.14
     array: abcdefghilmnopqrstuvz
colore RGB: verde
% g++ ./cpp/preprocessore-main.cpp -D LANG_EN -o ./out/esempio
% ./out/esempio
   boolean: 0
 character: C
   integer: 1234567890
   decimal: 3.14
     array: abcdefghilmnopqrstuvz
RGB color: green
```

È possibile eliminare una `#define` precedentemente assegnata per mezzo della direttiva `#undef`:

```
#define LANG_IT
#include "preprocessore-colore.h"
#undef LANG_IT

#ifdef LANG_IT
```

```
    #define  STR_BOOL   "booleano"
    #define  STR_CHAR   "carattere"
    #define  STR_INT    "intero"
    #define  STR_DEC    "decimale"
    #define  STR_ARRAY  "array"
    #define  STR_VERDE  "verde"
#else
    #define  STR_BOOL   "boolean"
    #define  STR_CHAR   "character"
    #define  STR_INT    "integer"
    #define  STR_DEC    "decimal"
    #define  STR_ARRAY  "array"
    #define  STR_VERDE  "green"
#endif
```

L'output di questo codice, sarà:

```
% g++ src/cpp/preprocessore-main.cpp -D LANG_EN -o src/out/esempio
% ./src/out/esempio
   boolean: 0
 character: C
   integer: 1234567890
   decimal: 3.14
     array: abcdefghilmnopqrstuvz
colore RGB: verde
```

Il preprocessore può essere sfruttato anche per creare delle *macro-istruzioni* che possano essere utilizzate con tipi diversi di dati.

```
#define MAGGIORE(a,b) ((a > b) ? a : b)
```

Quando il precompilatore trova una chiamata alla macro MAGGIORE, all'interno del codice, la sostituisce con l'istruzione corrispondente, rimpiazzando i parametri `a` e `b` con le variabili contenute nella chiamata.

```
/**
 * @file src/preprocessore-macro.cpp
 * Esempio di macro-istruzione del precompilatore.
 */

#include <iostream>

using namespace std;

/**
*  Definzione di una macro istruzione che
*  torna il maggiore fra due parametri.
```

```
*/
#define MAGGIORE(a,b) ((a > b) ? a : b)

int main ()
{
    /**
     *  Il codice di queste istruzioni è sostituito
     *  con quello associato alla macro:
     *
     *      cout << ((109 > 122) ? 109 : 122) << endl;
     *      cout << ((0.4 > 0.7) ? 0.4 : 0.7) << endl;
     *      cout << (('a' > 'z') ? 'a' : 'z') << endl;
    */
    cout << MAGGIORE(209, 122) << endl;
    cout << MAGGIORE(0.4, 0.7) << endl;
    cout << MAGGIORE('y', 'z') << endl;

    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/preprocessore-macro.cpp -o src/out/esempio
> src/out/esempio
209
0.7
z
```

Le macro del precompilatore sono eseguite prima che il codice sia compilato, quindi possono essere sfruttate anche per creare delle funzioni ex-novo.

```
/**
 * @file src/preprocessore-hashtag.cpp
 * Esempii di macro-istruzioni del precompilatore.
 */

#include <iostream>

using namespace std;

/** Definisce la macro-istruzione */
#define FUNZIONE(nome, parametro) int fnz_##nome() { return parametro; }

/**
*   Durante la precompilazone, questa macro
*   sarà sosituita dall'istruzione:
*
*     int fnz_macro(12) { return 12; }
```

```
*/
FUNZIONE(macro, 12)

/**
*   Definisce una macro che stampa un testo a video:
*/
#define OUTPUT(testo) cout << testo << endl;

/**
*   Definisce una macro che unisce la stringa "Pippo"
*   al parametro ricevuto in input:
*/
#define PIPPO "Pippo"
#define APPENDI(nome) PIPPO # nome

int main()
{
    /**
    *   La funzione fnz_macro ora non esiste, ma esisterà
    *   al termine della pre-compilazione del codice:
    */
    cout << fnz_macro() << endl;

    /** Stampa una stringa a video usando le altre due macro */
    OUTPUT(APPENDI(Pluto));
}
```

Questo esempio fa uso di due operatori proprii del preprocessore:

- l'operatore di *stringification* `#` (mi rifiuto di tradurre questo termine), che converte il parametro successivo in una stringa, aggiungendogli i doppi apici ed aggiungendo dei *backlslash* a eventuali doppi apici presenti nel parametro;
- l'operatore di *concatenazione* `##`, che unisce in un'unica stringa il parametro precedente e successivo.
  Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/preprocessore-hashtag.cpp -o src/out/esempio
> src/out/esempio
12
PippoPluto
```

Le macro-istruzioni del precompilatore sono uno strumento molto potente, ma devono essere utilizzate con oculatezza perché, essendo generate **prima** che dell'avvio della compilazione, non sono sottoposte ad alcun controllo di congruenza per il tipo dei parametri utilizzati. Questo, come vedremo in seguito, può generare degli errori molto subdoli e difficili da identificare.

Le tre forze che regolano la vita dell'Universo — Gravità, Entropia e Annosa Dicotomia — non influenzano solo la fisica, ma anche l'etica. Esistono valori che potremmo definire: "entropici" e valori "gravitazionali". I valori legati all'Entropia sono quelli che tendono a disgiungere e a esaltare il singolo rispetto alla massa, come la libertà, la ricchezza, il successo o la fama; i valori legati alla Gravità sono quelli che, al contrario, uniscono gli individui, come l'*égalité* e la *fraternité* dei Francesi, ma senza la ghigliottina.
Ieri abbiamo visto come i valori legati all'Entropia siano effimeri — non potrebbe essere altrimenti, per una forza che tende alla disgregazione — e di come chi li persegua sia spesso vittima dell'Annosa Dicotomia, ma questo non vuol dire che li si debba ripudiare. Non fare l'errore di pensare che ci sia una forza buona e una cattiva, come in *Star Wars*: sia la Gravità che l'Entropia sono necessarie per una corretta evoluzione dell'Universo. Il Maestro Canaro, che anche in tarda età amava molto andare in bicicletta, una volta mi disse che l'equilibrio dell'Universo è come l'equilibrio di un ciclista che percorra una strada sterrata in salita: non deve tenere il peso troppo in avanti, perché altrimenti la ruota posteriore perderebbe trazione e comincerebbe a slittare, ma non deve nemmeno tenere il peso troppo indietro, perché altrimenti la ruota anteriore si alleggerirebbe e non sarebbe più possibile sterzare.
Considerata la rispettabile quantità di cadute e contusioni collezionate dal Maestro Canaro nel corso delle sue escursioni in sella, ti consiglio di valutare solo il valore metaforico di questo insegnamento, ovvero che ci deve essere un bilanciamento fra Gravità ed Entropia, perché senza la Gravità, l'Universo è inutile, ma senza l'Entropia è noioso. In questo momento, l'Universo è in fase di espansione sotto l'influsso dell'Entropia ed è quindi normale che ci sia una preponderanza di azioni tendenti alla separazione. Per mantenere l'equilibrio, quindi, bisogna incentivare le attività e i valori che tendono a riunire, come l'amore o l'arte.
Molte persone pensano che amore e arte siano dei concetti vaghi e aleatorii, ma solo perché ne fraintendono l'essenza, ritenendoli dei concetti auto-esplicativi, che non occorre definire; qualcosa di simile alla definizione di "*pornografia*" che diede il Giudice Potter Steward, a proposito del film *Les Amants* di Louis Malle:

> I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. But I know it when I see it, and the motion picture involved in this case is not that.

Questo approccio entropico/soggettivo, per quanto corretto nel caso di *Les Amants*, è chiaramente sbagliato in termini generali: tutte le parole, anche quelle più comuni, possono essere interpretate in maniera differente da chi le ascolta o le utilizza. Così come la parola: *pesce*, per quanto banale, può non avere lo stesso valore per un biologo marino, un ecologista o per un pescatore, anche la parola *arte* può assumere significati diversi per un gallerista, per un artista o per un Papa.

È altrettanto sbagliata l'interpretazione entropico/romantica che comunemente si attribuisce alle parole: *amore* e *arte*. L'amore non è il sentimento vago che ha fatto la fortuna di poeti, musici e letterati, ma uno stato (più o meno persistente) del nostro organismo in cui riteniamo che qualcuno o qualcosa sia più importante di noi stessi. Di contro, l'Arte (ti prego di scrivere questo termine con l'iniziale maiuscola, nel tuo libro) non è l'esaltazione della personalità di un singolo, ma è — e dev'essere — la traccia del nostro cammino alla ricerca del Senso della Vita: Dio, per chi ci crede, o la Perfezione per i non credenti. Approfondiremo quest'ultimo concetto in séguito; qui e ora, come direbbe Céline Dion, *let's talk about love*.

Tutti noi agiamo in base a degli schemi di valori che determinano le nostre scelte. L'istinto di sopravvivenza, solitamente, ci spinge a porre la nostra persona al vertice di questa piramide, ma alle volte può capitarci di eleggere a nostro bene supremo qualcuno o qualcosa diverso da noi stessi. Quel senza Dio di Dawkins ha dimostrato come l'attaccamento che noi proviamo per i nostri parenti sia direttamente proporzionale al numero di cromosomi che condividiamo con essi e che quindi può essere riconducibile al desiderio primordiale di perpetuare il nostro patrimonio genetico. Questa interpretazione ribonucleica dell'amore funziona molto bene per i rapporti di sangue (padre/figlio, nonno/nipote, zio/nipote ecc.), ma non spiega l'amore fra mamma e papà o fra nonno e nonna — almeno, nelle famiglie che non ritengono l'incesto una pratica accettabile.
L'*egoismo dei geni* non spiega nemmeno altre forme d'amore come l'amor di Patria, che spinge i giovani a sacrificare la propria vita in guerra, l'amore per il prossimo, che porta il missionario a sacrificare la sua vita per aiutare i malati o l'amore per una forma di arte, uno sport o un lavoro. La realtà è che l'amore è una *backdoor*, o, meglio: una bomba a tempo, che viene inserita di nascosto nel tuo software, per essere certi che farai la cosa giusta quando arriverà il momento.
L'amore è l'unica forza allo stesso tempo gravitazionale ed entropica. Gravitazionale, perché unisce gli individui; entropica, perché li porta a riprodursi, replicando il loro DNA in qualcos'*altro*. Ti spinge a restare in casa, davanti a una tela, uno spartito o un foglio bianco o ti fa uscire, su una sella, una pista o con una macchina fotografica in mano. Ti porta in cima a una montagna o in una baraccopoli; al Polo o nel deserto; ti getta nel mare, ma non ti viene a salvare, come nella canzone; ti fa lasciare un lavoro sicuro perché non riconosci più la persona che vedi nello specchio la mattina o ti fa restare in un monastero, anche se.
E questa è la parte facile; poi c'è l'amore fra gli individui.
Ti sei mai innamorato? No? be', succederà. Ci sono due modi, di innamorarsi: o conosci una persona e te ne innamori o conosci una persona e scopri che è lei (o lui) che hai sempre amato. Il primo è il caso più comune; il secondo caso, non è un caso.
Il Maestro Canaro una volta disse (ma non credo fosse farina del suo sacco):

> Le donne cercano per tutta la vita il loro uomo ideale; nel frattempo, si sposano.

Questo è vero per tutti, uomini e donne. Il tuo DNA contiene i geni che hanno fatto nascere l'amore fra i tuoi genitori: occhi verdi, spalle larghe, un bel seno o magari un bel sedere. Allo stesso modo, dentro di te potrebbe esserci qualcosa che ti spingerà a cercare e ad amare una persona in particolare, perché quella persona è importante per la tua vita. Quando (se) la incontrerai, avrai l'impressione di conoscerla da sempre e capirai che tutte le volte che ti sei innamorato, ti sei innamorato di lei, anche se non la conoscevi ancora. Sfortunatamente, questo non vuol dire che la vostra storia durerà o che vivrete per sempre felici e contenti, come nelle favole. Anzi.
Il Cielo non è interessato alla tua felicità, ma al miglioramento, che, come ogni forma di crescita, ha il suo prezzo. Tutti sanno che il Maestro Canaro risolse il *Koan Mu* in una notte di profondo dolore dopo la morte di un cane di nome Lele. Molti, per ciò, ritengono il cane Lele un *Bodhisattva* che diede la vita per generare nel Maestro la prima scintilla dell'Illuminazione. Hanno ragione, ma solo in parte, perché la morte del cane Lele fu solo una delle cause dell'amarezza di quella notte. Il Maestro Canaro mi disse che buona parte del dolore era stato causato da una donna, che lui amava e dalla quale era stato ferito. Il motivo per cui me lo raccontò è lo stesso per cui io lo sto raccontando ora a te: anni dopo, quando il Maestro aveva già definito le basi del C'hi++, lui tornò da quella donna e la ringraziò per il torto che gli aveva fatto, perché lo aveva messo nella condizione d'animo necessaria a capire qualcosa a cui altrimenti avrebbe potuto non arrivare mai.
Ringraziare chi ci fa del bene con azioni gentili fa parte dell'educazione; ringraziare chi ci fa del bene attraverso il male non è facile, ma è necessario per far sì che quell'azione venga ripetuta anche nelle prossime esistenze.

La memoria

> cogito sum()

La gestione della memoria è la caratteristica principale del C++.

Come ti ho detto, tutti i linguaggi di programmazione sono un modo di vedere la memoria del computer. Quando tu *dichiari* una variabile con un'istruzione come:

```
signed int a = 1;
```

in realtà, stai dicendo al compilatore di prendere un'area di memoria di 32 bit (un `int` è grande 4 byte, ricordi?), associarle il nome `a` e scriverci dentro il valore binario:

```
00000000 00000000 00000000 00000001
```

Come puoi vedere, la maggior parte della variabile è inutilizzata, quindi, se hai a che fare con valori minori di 65.535, è meglio usare degli `short int`, che occupano solo due byte.
Allo stesso modo, se dichiari la stringa:

```
const char* motto = "Cogito ergo sum";
```

stai chiedendo al compilatore di prendere un'area di memoria di 16 byte, salvarne l'indirizzo iniziale nella variabile `motto` e poi scriverci dentro i 15 caratteri della frase più un ultimo carattere, con valore `0`, che indica la fine della stringa.
Puoi usare questo metodo se sai in anticipo quanto saranno grandi le variabili con cui avrai a che fare, ma se invece devi gestire dei valori di grandezza variabile (per esempio, l'input di un utente), hai due possibilità: o riservi preventivamente una quantità abbondante di spazio, o la allochi sul momento, in base alle tue necessità. Il primo caso va bene se devi gestire pochi dati di dimensioni ridotte, ma se devi gestire molte variabili di grandi dimensioni, l'allocazione dinamica, anche se più complessa da gestire, è più efficiente.
Il codice seguente è un esempio di gestione statica della memoria:

```
/**
 * @file memoria-statica.cpp
 * Gestione statica della memoria.
 */

#include <iostream>

/** Definisce la dimensione della stringa */
#define DIM_STRINGA 200

using namespace std;

int main(int argc, char** argv)
{
    /** Riserva uno spazio di 200 caratteri */
    char stringa[DIM_STRINGA];

    /**
    * Qui dovrebbe controllare che le stringhe ci siano
    * e che, sommate, siano più corte di 200 caratteri..
    */

    /** Copia le stringhe nell'area di memoria */
    sprintf(stringa, "%s %s", argv[1], argv[2]);

    /** Stampa la stringa a video */
    cout << stringa << endl;
```

```
    return 0;
}
```

Se lo compili e lo esegui, passando due stringhe come parametri, ottieni questo output:

```
% g++ src/cpp/memoria-statica.cpp -o src/out/esempio
% src/out/esempio "Stringa uno" "Stringa due"
Stringa uno Stringa due
```

Questo codice funziona se devi gestire solo due stringhe che, sommate, hanno meno di 200 caratteri: una condizione piuttosto restrittiva. Puoi aumentare il numero di stringhe e la dimensione dello spazio in memoria, ma otterresti un programma che è o sovra o sotto dimensionato. Questa non è buona programmazione. Il buon programmatore non deve sprecare risorse; per lui, il proverbiale bicchiere non è né mezzo pieno né mezzo vuoto: è grande il doppio del necessario.

```
/**
 * @file memoria-dinamica.cpp
 * Gestione dinamica della memoria.
 */

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    /**
    * Qui dovrebbe controllare che le stringhe ci siano..
    */

    /** Legge la lunghezza delle due stringhe */
    size_t len1 = strlen(argv[1]);
    size_t len2 = strlen(argv[2]);

    /**
     * Alloca un'area di memoria grande quanto le due
     * stringhe più un carattere di spazio e un
     * carattere terminatore.
     */
    size_t lunghezza = len1 + len2 + 2;
    char* stringa = new char[lunghezza];

    /** Copia le stringhe nell'area di memoria */
```

```
    sprintf(stringa, "%s %s", argv[1], argv[2]);

    /**
     * Mostra l'indirizzo dell'area di memoria,
     * la sua dimensione e il suo contenuto
     */
    cout << (void*)stringa << endl;
    cout << lunghezza      << endl;
    cout << stringa        << endl;

    /** Libera la memoria allocata */
    delete[] stringa;

    return 0;
}
```

L'output di questo programma, molto più parsimonioso della sua versione statica, è il seguente:

```
% src/out/esempio "Stringa uno" "Stringa due"
0x7fead1405910
24
Stringa uno Stringa due

% src/out/esempio "Stringa uno" "Stringa due più lunga"
0x7fbb6e405910
35
Stringa uno Stringa due più lunga

% src/out/esempio "Stringa uno" ""
0x7fcf8f405910
13
Stringa uno
```

La parola-chiave `new` permette di allocare una determinata quantità di memoria e torna un puntatore al primo indirizzo dell'area allocata:

```
char* stringa = new char[lunghezza];
```

Il puntatore `stringa`, adesso, contiene l'indirizzo dell'area di memoria allocata dall'istruzione `new`. Se lo passiamo come parametro all'operatore `<<`, dato che si tratta di un puntatore a `char`, verrà visualizzato come una stringa:

```
cout << stringa << endl;
```

Per conoscere il suo valore effettivo, dobbiamo convertirlo in un puntatore `void` con un'operazione di casting:

```
cout << (void*)stringa << endl;
```

Il lato negativo dell'allocazione dinamica della memoria è che, mentre un buffer statico "vive" quanto il blocco di istruzioni in cui è stato creato, la memoria allocata dinamicamente rimane occupata fino a che non viene liberata con un'istruzione `delete`:

```
delete[] stringa;
```

Il prossimo esempio dovrebbe aiutarti a capire come funziona la visibilità delle variabili nel C++:

```
/**
 * @file memoria-visibilita.cpp
 * Visibilità delle variabili.
 */

#include <iostream>

using namespace std;

/**
 * Le variabili definite all'esterno delle funzioni
 * si chiamano "variabili globali" e sono visibili
 * in tutto il codice.
 */
const char* stringa = "Stringa globale";

void funz()
{
    /**
     * Le variabili definite all'interno di una
     * funzione sono visibili solo all'interno
     * della funzione stessa.
     */
    const char* stringa = "Stringa esterna";

    /** Blocco di codice generico */
    {
        /**
         * Le variabili definite all'interno di un
         * blocco di codice sono visibili solo
         * all'interno del blocco di codice.
         */
        const char* stringa = "Stringa interna";

        /** Mostra la variabile interna */
        cout << "blocco: " << stringa << endl;
    }
```

```
    /**
     * Qui è visibile la variabile esterna.
     */
    cout << "funz:   " << stringa << endl;

    /**
     * L'operatore :: permette di accedere alla
     * variabile globale.
     */
    cout << "funz:   " << ::stringa << endl;

}

int main(int argc, char** argv)
{
    /** Richiama la funzione esterna */
    funz();

    /**
     * La funzione main non definisce una variabile
     * "stringa", quindi mostra la variabile globale.
     */
    cout << "main:   " << stringa << endl;

    return 0;
}
```

Se compili ed esegui questo codice, otterrai:

```
% g++ src/cpp/memoria-visibilita.cpp -o src/out/esempio
% src/out/esempio
blocco: Stringa interna
funz:   Stringa esterna
funz:   Stringa globale
main:   Stringa globale
```

Come vedi, la stringa definita globalmente è visibile sia nella funzione `main` che nella funzione `funz`; la variabile definita all'inizio della funzione `funz` è visibile all'interno della funzione stessa, ma non nel blocco di codice, dove è visibile la nuova variabile `stringa`. L' "aspettativa di vita" di ciascuna variabile dipende dal punto in cui è stata definita: la prima variabile esisterà per tutta la durata del programma; la seconda variabile viene creata quando si richiama la funzione `funz` e viene eliminata quando la funzione termina; la variabile all'interno del blocco di codice esiste solo per due istruzioni, poi il blocco di codice finisce e viene eliminata.
Al contrario, quando un'area di memoria è allocata dinamicamente, rimane

occupata fino a che il programma (o, più precisamente: il programmatore) non la rilascia con un'istruzione `delete`. Se il programma fa una sola chiamata, come nel nostro esempio, il fatto che una cinquantina di byte non siano disponibili per qualche minuto non crea grossi problemi (lo so per certo perché, nella prima versione dell'esempio, avevo dimenticato di aggiungere l'istruzione `delete` alla fine e il computer ha continuato a funzionare lo stesso), se però quello stesso programma continuasse a girare sullo stesso computer per lungo tempo, a poco a poco esaurirebbe tutta la memoria disponibile, causandone il blocco.
Per questo motivo, il linguaggio con la "J" ha un sistema di *garbage collection* che, come le squadre di pulizia dei Servizi Segreti, provvede a eliminare le prove dell'incompetenza dei suoi programmatori prima che questa arrechi danno ai sistemi. Qualcuno ti dirà che non è vero, che i programmatori "J" sono dei professionisti competenti, ma ragiona: se esiste un sistema di raccolta dei rifiuti, ci dovrà pur essere qualcuno che li produce, no?

Come le variabili del C++, anche gi esseri umani sono chiamati a vivere, svolgono il loro compito e alla fine vengono rimossi dal sistema. Le risorse che occupiamo sono un insieme di spazioni, invece che delle sequenze di byte, ma ciò che comunemente definiamo: *noi* non è altro che un modo di vedere le transizioni di energia all'interno del sistema. A differenza delle variabili del software, però, gli esseri umani sono capaci di valutare l'esito delle loro scelte o delle loro azioni e possono decidere se ripetere quella scelta o quell'azione in altri cicli di vita dell'Universo. Siamo, allo stesso tempo, una parte del programma, i *beta-tester* e gli sviluppatori e, se ci accorgiamo di un'istruzione errata, possiamo modificarla nelle successive "esecuzioni" del programma, così come un giocatore di scacchi evita di ripetere una mossa che si è rivelata perdente. In questo modo, genereremo a una nuova variante della storia che, come una variante scacchistica, potrà rivelarsi migliore o peggiore di quelle precedenti.
Un *Koan* molto famoso parla dello *Zen di ogni istante*:

> Gli studenti di Zen stanno con i loro maestri almeno dieci anni prima di presumere di poter insegnare a loro volta. Nan-in ricevette la visita di Tenno, che, dopo aver fatto il consueto tirocinio era diventato insegnante. Era un giorno piovoso, perciò Tenno portava zoccoli di legno e aveva con sé l'ombrello. Dopo averlo salutato, Nan-in disse: "Immagino che tu abbia lasciato gli zoccoli nell'anticamera. Vorrei sapere se hai messo l'ombrello alla destra o alla sinistra degli zoccoli". Tenno, sconcertato, non seppe rispondere subito. Si rese conto che non sapeva portare il suo Zen in ogni istante. Diventò allievo di Nan-in e studiò ancora sei anni per perfezionare il suo Zen di ogni istante.

Il Maestro Canaro detestava questo *Koan* perché era molto distratto. Se Nan-in l'avesse fatta a lui, la domanda, probabilmente l'avrebbe guardato con aria infastidita e gli avrebbe chiesto: "Quali zoccoli?". Malgrado ciò, ne riconosceva l'importanza perché più prestiamo attenzione a tutto ciò che facciamo, più errori

riusciremo a identificare e, si spera, a correggere.

Un'altra differenza fra gli esseri umani e il codice C++ è il nostro rapporto con l'operatore `delete`.
Gli uomini hanno sempre avuto paura della morte e, da Gilgamesh in poi, hanno sempre cercato di sfuggirle o di annullare il suo operato. Arte, religione, scienza: potremmo dire che tutto ciò che di buono abbiamo prodotto come specie nasce dal naturale rifiuto della transitorietà della nostra esistenza. La morte è stata la causa e, talvolta, il soggetto dei migliori frutti del nostro ingegno. Tutte queste creazioni, nel tempo, hanno creato un bozzolo culturale che, da un lato, ci ha protetto dalla paura e, dall'altro, ci ha dato degli ideali comuni per cui lottare, anche se non sempre a fin di bene. Negli ultimi due secoli, però, la Scienza ha squarciato questo bozzolo, privando gli esseri umani delle loro bugie confortanti e sostituendole con delle certezze inquietanti, mentre il suo figlio scemo (il Movimento del '68) ha spazzato via delle parti sicuramente rivedibili, ma fondamentali della nostra Società, come la famiglia o la scuola, senza darle nulla in cambio, tranne la minigonna.
C'è un aneddoto apocrifo sul Maestro Canaro che parla proprio di questo:

> Un giorno, la zia del Maestro Canaro lo chiamò e gli disse che il suo computer non funzionava più.
> «Si accende, ma non si connette a Internet e non riesce a spedire la posta elettronica. Ti prego, vieni ad aggiustarlo.»
> Il Maestro Canaro allora si recò dalla zia e le chiese di mostrargli il computer che non funzionava. La zia lo condusse nel suo studio e accese il PC portatile che era poggiato sulla scrivania. Come previsto, il computer si avviò regolarmente, ma quando il Maestro Canaro cercò di aprire il browser, comparve un messaggio di errore. Il Maestro allora provò ad aprire il programma per la posta elettronica, ma anche questa volta comparve un messaggio di errore.
> Sotto lo sguardo sempre più accorato di sua zia, il Maestro Canaro esaminò le impostazioni di rete, verificò che il modem funzionasse e fece un aggiornamento dei file di sistema, ma il problema rimase; così, il Maestro afferrò il computer e lo gettò dalla finestra.
> «Ma cosa hai fatto?!» esclamò la zia, esterrefatta.
> «Non funzionava, così l'ho buttato,» rispose il Maestro.
> «Ma non si poteva aggiustare?» chiese la zia, che ancora non riusciva a capacitarsi di ciò che era successo.
> «Forse sì, ma ci sarebbe voluto un sacco di tempo: meglio buttarlo via.»
> «E io adesso come lavoro, senza computer?» chiese la zia, attonita.
> Il Maestro Canaro si strinse nelle spalle.
> «Non ne ho idea, ma tanto non avresti potuto lavorare nemmeno con quello, perché era rotto.»
> «Lo so che era rotto!» esclamò la zia. «Ma quando una cosa neces-

saria è rotta, la si aggiusta, non la si butta via!»
Lungi dall'apparire contrito, il Maestro Canaro si alzò in piedi e, fissando negli occhi la zia, esclamò:
«E allora, voi, nel Sessantotto?!»

L'Umanità, messa di fronte all'ineluttabilità della fine e, allo stesso tempo, privata del conforto della religione e del sostegno della famiglia, ha reagito come un paziente a cui sia diagnosticato un male incurabile ed ha elaborato il suo dolore secondo le cinque fasi definite dalla dottoressa Kübler Ross:

Negazione: così come l'Epoca vittoriana aveva il tabù del sesso, la nostra "cultura" ha il tabù della morte: i nostri bis-nonni fingevano di non avere interessi carnali, noi fingiamo che la morte non esista. La neghiamo a parole, usando dei giri di parole per non nominarla: "Se n'è andato", "Non c'è più", "È scomparso", neanche si stesse parlando di un evaso o di un illusionista. La neghiamo nei fatti, isolando i moribondi nelle corsie d'ospedale, lontani dalle loro case e dai loro cari. La neghiamo nei nostri pensieri e nelle nostre azioni, perché altrimenti l'insensatezza delle nostre vite, spese inseguendo il miraggio effimero del successo diventerebbe evidente e innegabile.

Rabbia: anche se rabbia e aggressività sono sempre state presenti nella nostra storia, dalla metà del Secolo scorso, oltre che dall'oppressione e dal disagio, hanno cominciato a fiorire rigogliose anche dal benessere. Gli scontri fra *Mods* e *Rockers* negli anni '60, le lotte armate degli anni '70, il *Punk* e, in tempi più recenti, i *foreign fighters* e l'aggressività nei *social-network*: nessuno di questi fenomeni nasce nei ghetti o da uno stato di bisogno, sono tutti degli *hobby* del Ceto medio.

Negoziazione: nel 1982 Jane Fonda pubblicò una videocassetta nella quale insegnava a fare ginnastica aerobica nel salotto di casa a chiunque potesse permettersi di spendere sessanta Dollari per un VHS. Fu una delle videocassette più vendute di tutti i tempi e diede il via a una moda che divenne uno stile di vita per milioni di persone. La mania del *fitness*, insieme al rifiorire della spiritualità *New Age*, sono stati il modo in cui i popoli civilizzati hanno cercato di venire a patti con la nuova, terrificante realtà presentata loro dalla Scienza, prendendo atto dei proprii errori e cercando di porvi rimedio cambiando stile di vita. Sfortunatamente, però, gli pseudo-mistici non hanno mai capito che recitare il *Sutra del Loto* perché credi che possa aiutarti a realizzare i tuoi desiderii è una contraddizione in termini. La religione è come gli antibiotici: non fa effetto se non prendi la dose intera.

Depressione: la sindrome depressiva è la seconda malattia più diffusa al Mondo, dopo i disturbi cardiaci, segno evidente che la fase di negoziazione non ha sortito gli effetti sperati. Fallito ogni tentativo di combattere la realtà, l'Uomo del terzo Millennio ha cercato scampo nella fuga e si è rinchiuso in sé stesso come il Giappone dei Tokugawa, ma senza il rifiorire delle arti che si accompagnò al *Sakoku*, perché la Realtà non è un predatore, che possiamo sperare di seminare. La realtà è ovunque e anche isolandoci non possiamo sfuggirle; anzi: la solitudine

genera un *feedback* che amplifica lo stato di malessere e rende ancora più difficile e improbabile il raggiungimento dello stadio successivo, ovvero, la..

Accettazione: il solo modo per salvarsi è rinunciare agli insegnamenti dei cattivi maestri del passato e accettare il fatto di non essere delle gocce d'acqua uniche e insostituibili, ma di far parte del mare. Chi vuole, potrà credere che in quel mare ci sia Poseidone; gli altri saranno liberi di pensare che ci siano solo pesci, molluschi e alghe: non ha importanza; ciò che conta è sottrarsi all'influenza separatrice dell'Entropia e riacquistare un senso di appartenenza a qualcosa che va oltre le nostre brevi e limitate esistenze. Solo così, potremo vincere la nostra battaglia contro l'Annosa Dicotomia. Solo così, potremo smettere di inseguire una fama transeunte e trovare forza nella consapevolezza del fatto che ciascuno di noi può modificare l'evoluzione dell'Mondo con le sue azioni.
Secondo lo *Yoga Sutra Bhāṣya*:

> Il Mondo intero subisce una mutazione a ogni istante; così, tutte le qualità esteriori del Mondo dipendono da questo istante presente.

Così come tutti gli istanti sono importanti, per il Mondo, ciascun essere è importante per l'Universo, se svolge il compito che gli è stato assegnato; anche qualcuno o qualcosa che apparentemente è insignificante. È noto che il Maestro Canaro raggiunse il primo stato di Illuminazione quando vide un arbusto crescere su una parete di roccia a picco sul mare. Un arbusto apparentemente inutile: i suoi semi non avrebbero mai raggiunto una terra dove germogliare e i suoi rami erano troppo esili perché un uccello ci potesse costruire il suo nido; malgrado ciò, la caparbietà con cui quella pianta svolgeva il ruolo che gli era stato assegnato, permise al Maestro Canaro di capire ciò che molti libri non erano riusciti a spiegargli, posando la prima pietra di quello che sarebbe stato poi il suo insegnamento.
Quello che noi percepiamo come il nostro "valore" è relativo. Pensa alla variabile `a` che abbiamo visto all'inizio:

```
00000000 00000000 00000000 00000001
```

Il bit iniziale vale `0`, che è un valore nullo, se lo consideri individualmente, ma se invece lo valuti in rapporto ai due byte a cui appartiene, diventa il valore da cui dipende il segno della variabile. Se per un caso il valore di quel bit diventasse `1`, il valore della variabile `a` diventerebbe negativo, con delle ripercussioni imprevedibili sul programma. Molte, troppe persone sono come quel bit e pensano di valere zero perché considerano il proprio valore solo in termini soggettivi. Mesmerizzati dall'Annosa Dicotomia, spendono tutte le loro energie cercando di valere `1` e si distraggono così da quello che sarebbe stato il loro destino, con conseguenze molto più gravi di un'alterazione di segno in una variabile.

Il buon programmatore deve sempre controllare che il suo codice non contenga dei difetti e ciò che ti ho appena detto ne ha uno, piuttosto grave: può essere utilizzato come pretesto per creare un sistema di caste che limiti le possibilità di

crescita di coloro che effettivamente sono destinati ad aumentare il loro valore. La domanda quindi è: come può, un bit o un essere senziente, decidere quale sia il suo valore esatto? Sicuramente non può dirglielo un'Autorità costituita né tanto meno una consuetudine. La famiglia o la scuola possono dargli dei suggerimenti, ma non possono decidere per lui: genitori e maestri hanno sicuramente più esperienza, ma non sono infallibili. La risposta è in una frase di Jacopone da Todi:

> Prima devi sape' perché stai ar monno.
> Quanno sai er perché, te devi impara' a stacce .

In queste due frasi è racchiuso il senso della Vita: capire quale sia il proprio ruolo e svolgerlo al meglio delle proprie possibilità, senza farsi influenzare dalle mode e soprattutto senza cadere vittima dell'Annosa Dicotomia.
Il C++ è un linguaggio a tipizzazione forte, perciò, quando si dichiara una variabile, le si deve sempre assegnare un tipo di dato:

```
bool    booleano  = false;
char    carattere = 'C';
int     intero    = 1234567890;
float   decimale  = 3.14;
char    array[]   = "abcdefghilmnopqrstuvz";
```

Allo stesso modo, tutto ciò che esiste ha delle capacità che sono funzionali al suo ruolo nell'Universo. Suonare, scrivere, recitare, insegnare, convincere, guidare, amare: a ciascuno di noi, l'Universo dà uno strumento e un banco di lavoro, anche se il ruolo che ci verrà assegnato non sempre è evidente fin dalla nascita. Prendi Lech Walesa, per esempio (è l'unico Nobel per la Pace che mi viene in mente che non fosse laureato): lui cominciò a lavorare come elettricista navale, ma le sue capacità lo trasformarono in un leader politico. Oppure, meglio, pensa a te stesso: se avessi voluto, a diciott'anni avresti potuto lasciare l'Ordine, tornare a casa e diventare un *vice-qualcosa* nell'azienda della tua famiglia, invece sei rimasto qui a scrivere il tuo libro. Sarà stata la scelta giusta? Cosa si aspettava, l'Universo, da te? che, come monaco, rendessi il Mondo un luogo migliore con la tua Bibbia per smanettoni o che, come manager, migliorassi le condizioni di vita dei dipendenti della tua azienda? Lo scopriremo solo quando torneremo a essere Uno; fino ad allora, potremo solo fare delle supposizioni. Se sei rimasto qui per pigrizia o perché avevi paura del Mondo o della tua famiglia, hai fatto un errore. Se sei rimasto perché non desideri il lusso o il potere, potresti aver fatto la scelta giusta. Forse. Se sei rimasto, anche se sarebbe stato più facile e comodo tornare in seno alla tua famiglia, perché pensavi che restando qui saresti stato più utile, o sei terribilmente ingenuo o hai fatto la cosa giusta.
A ogni modo, bisogna sempre fare bene il proprio mestiere, quale che sia, perché non sappiamo come e quando dovremo renderci utili. Una volta, un amico del Maestro Canaro, appena tornato dal Giappone, gli disse:

> La differenza fra noi e i Giapponesi è che se tu, qui, metti uno a

> pulire i cessi, quello si sente disprezzato e lavora male, mentre un giapponese cerca di diventare il miglior pulitore di cessi di tutto il Paese.

Un altro modo per capire se si è nel giusto, è in base all'amore. L'amore è la droga perfetta: dà assuefazione come il *crack* e ti fa credere di saper volare, come l'LDS; è il miele con cui la Vita attira le formiche dei nostri pensieri, facendole andare dove vuole lei. Quindi, se quello che ti spinge ad agire è una qualche forma di amore — e non solo amore per qualcuno, ma anche per qualcosa, che sia il mare, la montagna, uno sport, una forma di arte o un lavoro — è molto probabile che tu stia facendo ciò che devi.
Nell'ottica della gestione della memoria, l'amore ha anche un altro utilizzo, riassunto nel precetto:

> Amiamo ciò che ci ucciderà (se tutto va bene)

Adesso, però, è tardi e questa lezione è stata fin troppo lunga; ne parleremo un'altra volta.

Le funzioni

> Call me: IsNull

Le funzioni sono la caratteristica principale del C++.

Lo scopo delle funzioni è di migliorare la gestione del codice. È possibile scrivere un programma che abbia solo la funzione `main`, ma questo ha un senso solo per programmi molto semplici, come alcuni degli esempii che abbiamo visto finora. Il flusso di un programma *non banale*, come direbbe Stroutsup, sarà sempre suddiviso in più funzioni perché in questo modo il codice sarà più facile da leggere, da correggere e da modificare.

```
/**
 * @file tipi-di-dato-dimensione.cpp
 * Mostra la dimensione dei principali tipi di dato del C++.
 */

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int w = 12;
```

```
    cout << setw(w) << "bool: "   << sizeof(bool)   << endl;
    cout << setw(w) << "char: "   << sizeof(char)   << endl;
    cout << setw(w) << "int: "    << sizeof(int)    << endl;
    cout << setw(w) << "float: "  << sizeof(float)  << endl;
    cout << setw(w) << "double: " << sizeof(double) << endl;

    return 0;
}
```

Questo è il codice di un esempio che abbiamo visto nella lezione sui tipi di dato. Se decidessimo di modificare il modo in cui il programma mostra i dati all'utente, dovremmo intervenire su tutte le righe del programma, cosa che sarebbe noiosa e potrebbe generare degli errori. Al contrario, se isoliamo la funzione di visualizzazione, le eventuali modifiche o correzioni dovranno essere applicate solo in un punto. Il codice dell'esempio successivo è un esempio, perfettibile, di questo approccio:

```
/**
 * @file tipi-di-dato-limiti.cpp
 * Mostra i valori possibili per i principali tipi di dato del C++.
 */

#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

/**
 * Funzione template che mostra la dimensione e i
 * valori minimi e massimi per un tipo di dato.
 */
template<typename T>
void dimensione()
{
    cout << sizeof(T) << " bytes,"
         << setw(4)   << "da:"
         << setw(21)  << numeric_limits<T>::min()
         << setw(4)   << "a:"
         << setw(21)  << numeric_limits<T>::max()
         << endl;
}

int main()
{
    /** Variazione della dimensione e dei valori del tipo int */
```

```
    cout << setw(16) << "int: ";
    dimensione<int>();
    cout << setw(16) << "unsigned int: ";
    dimensione<unsigned int>();
    cout << setw(16) << "short int: ";
    dimensione<short int>();
    cout << setw(16) << "unsigned short: ";
    dimensione<unsigned short>();
    cout << setw(16) << "long int: ";
    dimensione<long int>();
    cout << setw(16) << "unsigned long: ";
    dimensione<unsigned long>();

    return 0;
}
```

Ogni volta che nel tuo codice ci sono delle istruzioni che si ripetono, valuta la possibilità di isolarle in una funzione. Per esempio, nella funzione `main` del programma qui sopra ci sono delle funzioni di output ripetute, cosa che complica la lettura del codice, ne rende laboriosa la modifica e aumenta la probabilità di fare degli errori, perché sei tu che definisci la stringa con il nome del tipo di dato e devi fare attenzione a scrivere il valore giusto ogni volta:

```
cout << setw(16) << "int: ";
...
cout << setw(16) << "unsigned int: ";
...
cout << setw(16) << "short int: ";
...
cout << setw(16) << "unsigned short: ";
...
cout << setw(16) << "long int: ";
...
cout << setw(16) << "unsigned long: ";
```

Nel caso specifico, si trattava di una scelta inevitabile perché, essendo uno dei primi esempii che ti ho fatto, non volevo complicarti troppo le idee, ma non è questo il modo corretto di scrivere codice. È sbagliato (o, quanto meno, rischioso) mischiare elaborazione dei dati e funzioni di interfaccia, specie in questo caso, dove parte dell'output è già demandato a una funzione specifica. Meglio lasciare che sia la funzione `dimensione` a gestire tutto l'output, mentre la funzione `main` si limiterà a definire il flusso dell'elaborazione, richiamando la funzione di output per i tipi di dato che ci interessano:

```
/**
 * @file funzioni-limiti-modificato.cpp
 * Esempio sui limiti del C++ con output in un'unica funzione.
 */
```

```
#include <iostream>
#include <iomanip>
#include <limits>
#include <typeinfo>
#include <cxxabi.h>

using namespace std;

/**
 * Mostra il nome, la dimensione e i valori minimi
 * e massimi per il tipo di dato corrente.
 */
template<typename T>
void dimensione()
{
    cout << setw(14)
         << abi::__cxa_demangle(typeid(T).name(), NULL, NULL, NULL)
         << ": "
         << sizeof(T) << " bytes,"
         << setw(4)   << "da:"
         << setw(21)  << numeric_limits<T>::min()
         << setw(4)   << "a:"
         << setw(21)  << numeric_limits<T>::max()
         << endl;
}

/**
 * Il nuovo codice della funzione main:  più compatto
 * e più facile da leggere o modificare.
 */
int main(int argc, char** argv)
{
    dimensione<int>();
    dimensione<short int>();
    dimensione<unsigned short>();
    dimensione<long int>();
    dimensione<unsigned long>();
    return 0;
}
```

L'istruzione:

```
abi::__cxa_demangle(typeid(T).name(), NULL, NULL, NULL);
```

mostra il tipo di dato della variabile corrente. Per il momento, accettala in maniera dogmatica; ti spiegherò il comportamento dell'operatore `::` quando

parleremo delle classi.
L'output di questo codice è identico a quello dell'esempio precedente; anzi: è un po' meglio, perché ho ridotto la spaziatura della prima colonna da 16 a 14 caratteri e l'ho fatto modificando solo un'istruzione `setw(14)`, contro le sei del codice precedente:

```
% g++ src/cpp/funzioni-limiti-modificato.cpp -o src/out/esempio
% src/out/esempio
           int: 4 bytes, da:          -2147483648  a:           2147483647
         short: 2 bytes, da:                -32768  a:                32767
unsigned short: 2 bytes, da:                     0  a:                65535
          long: 8 bytes, da: -9223372036854775808  a:  9223372036854775807
 unsigned long: 8 bytes, da:                     0  a: 18446744073709551615
```

Le funzioni sono uno dei tipi di dato del C++ e, come tutti i tipi di dato, possono essere *dichiarate* e *definite*.
La dichiarazione di una funzione stabilisce il suo tipo di ritorno e i parametri richiesti in input:

```
float scorporaIVA(long stipendio, float aliquota);
```

La definizione di una funzione, ne stabilisce il comportamento:

```
float scorporaIVA(long stipendio, float aliquota)
{
    float valore;
    valore = stipendio / ((100 + aliquota) / 100);
    return valore;
}
```

Come ti ho detto quando abbiamo parlato dei linguaggi di programmazione, la generazione di un file eseguibile avviene in due fasi: per prima cosa il compilatore converte il codice C++ in un *file oggetto*, poi il *linker* trasforma i file oggetto (potrebbero essere più d'uno) in un unico eseguibile. Perché questo processo possa funzionare, la *dichiarazione* di una funzione deve essere presente in tutti i brani di codice che la utilizzano, per consentire al compilatore di controllare che l'utilizzo che se ne fa sia corretto; la *definizione*, al contrario, deve comparire solo una volta.

```
/**
 * @file funzioni-stipendio-main.cpp
 * Funzione main del programma con file oggetto separati.
 */
#include <iostream>

/**
 * Dichiarazione della funzione raddoppiaStipendio.
 * L'ho messa qui per semplicità: di solito, le dichiarazioni
```

```
 * si trovano in un file di include separato.
 */
float raddoppiaStipendio(float stipendio);

/**
 * La funzione main richiama la funzione raddoppiaStipendio,
 * che è definita in un altro file sorgente.
 */
int main(int argc, char** argv)
{
    std::cout << raddoppiaStipendio(1500) << std::endl;
    return 0;
}
```

```
/**
 * @file funzioni-stipendio-funz.cpp
 * Funzione del programma con file oggetto separati.
 */

/**
 * Definizione della funzione raddoppiaStipendio.
 */
float raddoppiaStipendio(float stipendio)
{
    return stipendio * 2;
}
```

Se compiliamo separatamente i due file qui sopra, aggiungendo il parametro `-c`, che dice al compilatore di generare solo il file oggetto senza richiamare il linker per la generazione di un eseguibile:

```
% g++ -c src/cpp/funzioni-stipendio-main.cpp -o src/out/main.o
% g++ -c src/cpp/funzioni-stipendio-funz.cpp -o src/out/funz.o
```

e poi generiamo un file eseguibile utilizzando i due file oggetto:

```
% g++ -o src/out/esempio src/out/main.o src/out/funz.o
```

otterremo tre file, due *object-file* e il file eseguibile `esempio`, che darà il risultato atteso:

```
% ls -1 src/out
esempio
funz.o
main.o

% src/out/esempio
3000
```

Ma siccome io sono pigro, negli esempii che ti farò, utilizzerò sempre un comando unico per la compilazione e il *linking* dei programmi:

```
g++ src/cpp/funzioni-stipendio-funz.cpp  \
    src/cpp/funzioni-stipendio-main.cpp  \
    -o src/out/esempio
% src/out/esempio
3000
```

Se non dichiarassimo la funzione `raddoppiaStipendio` nel file che contiene la funzione `main`, il compilatore ci darebbe l'errore:

```
src/cpp/funzioni-stipendio-main.cpp:19:18: error: use of undeclared identifier 'raddoppiaSti
    std::cout << raddoppiaStipendio(1500) << std::endl;                      ^
1 error generated.
```

Otterremmo lo stesso errore se definissimo una funzione dopo che un altra parte del programma l'ha richiamata:

```
/**
 * @file funzioni-stipendio-errore.cpp
 * Funzione che genera un errore di compilazione.
 */
#include <iostream>

/**
 * La funzione main richiama la funzione raddoppiaStipendio,
 * ma il compilatore ancora non sa che esiste: errore.
 */
int main(int argc, char** argv)
{
    std::cout << raddoppiaStipendio(1500) << std::endl;
    return 0;
}

/**
 * Definizione della funzione raddoppiaStipendio DOPO
 * il suo utilizzo da parte della funzione main.
 */
float raddoppiaStipendio(float stipendio)
{
    return stipendio * 2;
}
```

```
g++ -c src/cpp/funzioni-stipendio errore.cpp

src/cpp/funzioni-stipendio-errore.cpp:13:18: error: use of undeclared identifier 'raddoppiaS
    std::cout << raddoppiaStipendio(1500) << std::endl;
                 ^
```

```
1 error generated.
```

Quando si scrive un programma in un unico file sorgente, o si definiscono le diverse funzioni prima che vengano utilizzate, mettendo la funzione `main` in fondo, oppure le si deve dichiarare all'inizio del file. (In realtà, non occorre metterle tutte all'inizio del file, basta che la dichiarazione preceda l'utilizzo, ma è più scomodo: mettile all'inizio.)
Avresti ottenuto un errore di compilazione anche se avessi provato a separare in due file distinti la funzione `main` e la funzione `dimensione` dell'esempio qui sopra, anche se avessi dichiarato `dimensione` prima del suo utilizzo nella funzione `main`:

```
% g++ src/cpp/funzioni-limiti-main.cpp -c -o src/out/main.o
src/cpp/funzioni-limiti-main.cpp:19:5: error: use of undeclared identifier 'dimensione'
    dimensione<int>();
    ^
src/cpp/funzioni-limiti-main.cpp:19:19: error: expected '(' for function-style cast or type
    dimensione<int>();
               ~~~^
src/cpp/funzioni-limiti-main.cpp:19:21: error: expected expression
    dimensione<int>();
                    ^
src/cpp/funzioni-limiti-main.cpp:20:5: error: use of undeclared identifier 'dimensione'
    dimensione<short int>();
    ^
src/cpp/funzioni-limiti-main.cpp:20:22: error: expected '(' for function-style cast or type
    dimensione<short int>();
               ~~~~~ ^
src/cpp/funzioni-limiti-main.cpp:21:5: error: use of undeclared identifier 'dimensione'
    dimensione<unsigned short>();
    ^
src/cpp/funzioni-limiti-main.cpp:21:25: error: expected '(' for function-style cast or type
    dimensione<unsigned short>();
               ~~~~~~~~ ^
src/cpp/funzioni-limiti-main.cpp:22:5: error: use of undeclared identifier 'dimensione'
    dimensione<long int>();
    ^
src/cpp/funzioni-limiti-main.cpp:22:21: error: expected '(' for function-style cast or type
    dimensione<long int>();
               ~~~~ ^
src/cpp/funzioni-limiti-main.cpp:23:5: error: use of undeclared identifier 'dimensione'
    dimensione<unsigned long>();
    ^
src/cpp/funzioni-limiti-main.cpp:23:25: error: expected '(' for function-style cast or type
    dimensione<unsigned long>();
               ~~~~~~~~ ^
```

Questo avviene perché i `template` non sono vere funzioni, ma solo degli sche-

mi che il compilatore utilizza per generare la versione corretta del codice. La dichiarazione delle funzioni template, quindi deve comprendere anche la loro definizione, per dare modo al compilatore di gestire appropriatamente la chiamata.

Prima ti ho detto di *valutare* la possibilità di isolare in una funzione le istruzioni che si ripetono all'interno del tuo codice, perché non sempre creare una funzione è la scelta corretta.
Il software, come molte attività umane, è il frutto di una serie di compromessi e tu dovrai fare scelte architetturali differenti a seconda del tipo di programma che devi realizzare. Un buon software, oltre che funzionare correttamente, dovrebbe essere veloce, facile da modificare e richiedere poche risorse di sistema. Alle volte, però, si deve sacrificare una di queste caratteristiche positive per esaltarne un'altra. Per esempio, se dovessi scrivere un software estremamente veloce, potrebbe essere meglio avere delle istruzioni duplicate che delle chiamate a funzione, perché richiamare una funzione causa inevitabilmente dei rallentamenti. Però, se replichi delle istruzioni, non solo aumenti le dimensioni del programma, ma lo rendi anche più difficile da leggere e da modificare, che è male.
In questi casi, l'aumento delle dimensioni del codice è inevitabile, ma le *funzioni inline* ti permettono di mantenere il codice leggibile e modificabile:

```
/**
 * @file funzioni-inline-1.cpp
 * Esempio di codice con istruzioni ripetute.
 */

#include <iostream>
#include <iomanip>
#include <fstream>

#define S_FILENAME  "src/out/esempio.txt"

using namespace std;

int main(int argc, char** argv)
{
    /**
     * Crea una variabile di classe ofstraeam
     * per gestire il file di output  */
    ofstream doc;

    /** Apre il file di output */
    doc.open (S_FILENAME);
    cout << "ho aperto il file " << endl;
```

```
    /** Scrive sul file di output */
    doc << "Testo del documento.\n";
    cout << "ho scritto sul file" << endl;

    /** Chiude il file di output */
    doc.close();
    cout << "ho chiuso il file" << endl;

    return 0;
}
```

Compilando ed eseguendo il codice qui sopra, ottieni:

```
% g++ src/cpp/funzioni-inline-1.cpp -o src/out/esempio
% ./src/out/esempio
ho aperto il file
ho scritto sul file
ho chiuso il file
```

Le istruzioni di output nell'esempio si differenziano solo per il testo da visualizzare e potrebbero benissimo essere isolate in una funzione autonoma.

```
/**
 * @file funzioni-inline-2.cpp
 * Esempio di codice con funzione di output unica.
 */

#include <iostream>
#include <iomanip>
#include <fstream>

#define S_FILENAME  "src/out/esempio.txt"

using namespace std;

/** Funzione di output unica */
void log(const char* messaggio)
{
    /** Mostra il messaggio a video */
    cout << messaggio << endl;
}

int main(int argc, char** argv)
{
    /** Crea una variabile per gestire il file di output */
    ofstream doc;
```

```
    /** Apre il file di output */
    doc.open (S_FILENAME);
    log("ho aperto il file");

    /** Scrive sul file di output */
    doc << "Testo del documento.\n";
    log("ho scritto sul file");

    /** Chiude il file di output */
    doc.close();
    log("ho scritto il file");

    return 0;
}
```

L'output di questo programma è identico a quello dell'esempio precedente, ma se aggiungi la parola chiave `inline` prima del tipo di ritorno della funzione `log`:

```
inline void log(const char* messaggio)
{
    /** Mostra il messaggio a video */
    cout << messaggio << endl;
}
```

e compili nuovamente il programma, vedrai che la dimensione del file eseguibile è aumentata, perché il compilatore ha sostituito tutte le chiamate a funzione con una copia del codice della funzione stessa. Anche se le dimensioni dell'eseguibile sono aumentate, il codice è ancora facilmente leggibile e modificabile:

```
/**
 * @file funzioni-inline-3.cpp
 * Esempio di funzione inline.
 */

#include <iostream>
#include <iomanip>
#include <fstream>

#define LOG_DEBUG   1
#define LOG_AVVISO  2
#define LOG_ERRORE  3

#define S_DEBUG     "debug"
#define S_AVVISO    "avviso"
#define S_ERRORE    "errore"

#define ERR_NO_FILE_NAME  -1
#define ERR_NO_FILE_OPEN  -2
```

```
using namespace std;

/** Funzione di output inline */
inline void log(const char* messaggio, int livello)
{
    /** Definisce la spaziatura del primo campo */
    cerr << setw(7);

    /** Scrive il livello del messaggio */
    switch(livello) {
        case LOG_DEBUG:  cerr << S_DEBUG ; break;
        case LOG_AVVISO: cerr << S_AVVISO; break;
        default:         cerr << S_ERRORE; break;
    }

    /** Scrive il testo del messaggio */
    cerr << ": " << messaggio << endl;
}

int main(int argc, char** argv)
{
    /** Crea una variabile per gestire il file di output */
    ofstream doc;

    /** Puntatore per il nome del file di output */
    const char* filename = NULL;

    /** Se manca il nome del file di output, errore */
    if(argc < 2) {
        log("specificare il path del file", LOG_ERRORE);
        return ERR_NO_FILE_NAME;
    }

    /** Legge il nome del file di output */
    filename = argv[1];

    /** Prova ad aprire il file di output */
    doc.open (argv[1]);

    /** Se c'è stato un errore, lo segnala ed esce */
    if(!doc.is_open()) {
        log("impossibile aprire il file", LOG_ERRORE);
        return ERR_NO_FILE_OPEN;
    }
```

```
    /** OK, il file è pronto */
    log("ho aperto il file", LOG_AVVISO);

    /** Scrive sul file di output */
    doc << "Testo del documento.\n";
    log("ho scritto sul file", LOG_DEBUG);

    /** Chiude il file di output */
    doc.close();
    log("ho chiuso il file", LOG_AVVISO);

    return 0;
}
```

Se compili il codice e lo esegui, ottieni:

```
% g++ src/cpp/funzioni-inline-3.cpp -o src/out/esempio

% src/out/esempio
 errore: specificare il path del file

% src/out/esempio /esempio.txt
 errore: impossibile aprire il file

% src/out/esempio src/out/esempio.txt
 avviso: ho aperto il file
  debug: ho scritto sul file
 avviso: ho chiuso il file
```

Per la cronaca: nel secondo caso, il programma fallisce perché l'utente non ha privilegi di scrittura nella *root-directory*.

L'esistenza di una funzione `inline` ricorda un po' un passo del *Samyutta Nikaya*:

> Non esiste distinzione fra un essere, la sua funzione e il tempo della sua apparizione. Gli esseri appaiono dalla non-esistenza, esistono per un istante e poi cessano di esistere. La loro esistenza, attività e azione sono un'unica cosa. Passato e futuro sono meri nomi.

In generale, comunque, il ciclo di vita di una funzione prevede tre fasi distinte: la *dichiarazione*, la *definizione* e il suo successivo utilizzo:

```
/** Dichiarazione */
float raddoppiaStipendio(float stipendio);

/** Definizione */
float raddoppiaStipendio(float stipendio)
{
```

```
    return stipendio * 2;
}

/** Chiamata */
float importo = raddoppiaStipendio(1500);
```

Solo la funzione `main` fa eccezione a questa regola, perché non richiede una dichiarazione e non viene richiamata da altre funzioni. La variabile `float stipendio`, che compare fra parentesi sia nella dichiarazione che nella definizione della funzione, viene detta: *parametro* della funzione. Il valore `1500` che viene attribuito al parametro nella chiamata a funzione, è detto: *argomento*. Queste variabili sono chiamate anche: *parametri formali*, nel caso della dichiarazione e della definizione e: *parametri attuali*, nel caso della chiamata.
Quando richiami una funzione, le puoi passare gli argomenti in tre modi: per *valore*, per *riferimento*, per *puntatore*:

```
/**
 * @file funzioni-argomenti.cpp
 * Esempio di gestione degli argomenti di una funzione.
 */

#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

/**
*   Chiamata per valore: tutto quello che succede qui
*   alle variabili a e b non influenza le variabili
*   x e y della funzione chiamante.
*/
void perValore(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
    return;
}

/**
*   Chiamata per reference: tutto quello che succede qui
*   alle variabili a e b è come se fosse fatto alle variabili
*   x e y della funzione chiamante.
*/
void perRiferimento(int &a, int &b)
{
```

```
    int tmp = a;
    a = b;
    b = tmp;
    return;
}

/**
*   Chiamata per puntatore: anche qui, quello che succede
*   alle variabili a e b avviene anche alle variabili x e y
*   della funzione chiamante, ma il codice è meno chiaro.
*/
void perPuntatore(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main(int argc, char** argv)
{
    int  x = 11; int  y = 22;
    cout << setw(20) << "Valore iniziale: x=" << x << ", y=" << y << endl;

    int* p = &x;
    int& r = x;

    cout << "p:" << p << ",r=" << r << endl;

    /**
    *   Passaggio per valore: l'argomento della chiamata
    *   a funzione è il valore delle variabili x e y:
    */
    perValore(x,y);
    cout << setw(20) << "perValore: x=" << x << ", y=" << y << endl;

    /**
    *   Passaggio per reference: l'argomento della chiamata
    *   a funzione sono le variabili x e y, ma la funzione
    *   utilizzerà dei loro alias, non il loro valore:
    */
    perRiferimento(x,y);
    cout << setw(20) << "perRiferimento: x=" << x << ", y=" << y << endl;

    /**
    *   Passaggio per puntatore: l'argomento della chiamata
```

```
    *   è esplicitamente l'indirizzo delle variabili x e y:
    */
    perPuntatore(&x,&y);
    cout << setw(20) << "perPuntatore: x=" << x << ", y=" << y << endl;

    return 0;
}
```

L'output di questo programma è:

```
% g++ src/cpp/funzioni-argomenti.cpp -o src/out/esempio
% src/out/esempio
Valore iniziale: x=11, y=22
      perValore: x=11, y=22
 perRiferimento: x=22, y=11
   perPuntatore: x=11, y=22
```

Passare la *reference* a una variabile come argomento di una funzione equivale a passarle la variabile stessa. Questo può essere un bene nel caso di funzioni che richiedano in input oggetti di grosse dimensioni o che abbiano la necessità di modificare direttamente il valore delle variabili passategli come parametri, ma va evitato in qualsiasi altro caso, perché permette al programma di modificare il valore di una variabile in maniera subdola, che può essere molto difficile da scoprire in caso di errori.
Un'altra cosa da sapere, a proposito dei parametri delle funzioni, è che possono avere dei valori di default:

```
/**
 * @file _man/src/funzioni-parametri-default.cpp
 * Esempio di funzione con parametri di default.
 */

#include <iostream>

using namespace std;

/** Funzione con parametri di default  */
void stampaValore(int valore = 0)
{
    /** Stampa a video il valore del parametro */
    cout << "Valore: " << valore << endl;
}

int main(int argc, char** argv)
{
    /** Richiama la funzione con un argomento */
    stampaValore(123);
```

```
    /** Richiama la funzione senza argomenti */
    stampaValore();

    return 0;
}
```

L'output di questo programma sarà:

```
% g++ src/cpp/funzioni-parametri-default.cpp -o src/out/esempio
% src/out/esempio
Valore: 123
Valore: 0
```

Il valore di default del parametro deve essere indicato nella dichiarazione della funzione. Ricordati però che, quando si assegna un valore di default a uno dei parametri di una funzione, bisogna fare altrettanto con tutti i parametri che lo seguono, se ce ne sono:

```
void funz1(float f, void * ptr = NULL);        // OK
void funz2(double d, int b = 2, char c = 'c'); // OK
void funz3(int i = 3, int n );                 // ERRORE!
```

I valori di default si utilizzano quando a uno o più parametri della funzione è assegnato spesso un determinato valore. Per esempio, se la funzione `log` che abbiamo visto nell'esempio precedente fosse richiamata prevalentemente con uno stesso valore per il parametro `livello`, glielo si potrebbe assegnare come default:

```
inline void log(const char* messaggio, int livello = LOG_AVVISO);
```

rendendo la scrittura del codice più facile e veloce:

```
log("ho aperto il file");

doc << "Testo del documento.\n";
log("ho scritto sul file", LOG_DEBUG);

doc.close();
log("ho chiuso il file");
```

In una delle nostre prime chiacchierate, ti ho detto che il Buon Programmatore, mentre scrive il codice, si chiede sempre se possa esistere sistema più efficiente di fare ciò che sta facendo. Oggi ne hai avuto una dimostrazione: abbiamo migliorato il primo esempio del paragrafo sulle funzioni `inline` unificando la gestione dei messaggi in un'unica funzione e poi l'abbiamo migliorato ancora rendendo quella funzione `inline`. C'è un problema, però: l'output del programma:

```
% ./src/out/esempio
ho aperto il file
ho scritto sul file
```

```
ho chiuso il file
```

va bene solo se il file da gestire è uno solo, come nel nostro caso. Se però ci fossero due (o più) file di input o di output, sarebbe utile sapere a *quale* file si riferisca il messaggio. Per risolvere il problema, potremmo aumentare il numero di parametri formali della funzione `log`:

```
void log(int livello, const char* messaggio, const char* file);
```

ma il nuovo parametro sarebbe inutile nel caso di chiamate come:

```
log(LOG_ERRORE, "specificare il path del file");
```

In alternativa, potremmo definire il messaggio all'interno della funzione chiamante:

```
string s1 = "ho chiuso il file: ";
string s2 = filename;
string s3 = s1 + s2;
log(LOG_AVVISO, s3.c_str());
```

ma questo renderebbe il codice più pesante, più complicato e più lento; inoltre, trasferirebbe nelle funzioni chiamanti parte delle funzionalità di output che avevamo felicemente isolato nella funzione `log`.
La soluzione corretta per questo tipo di problemi sono le *funzioni con parametri variabili*.
Un esempio tipico di questo tipo di funzioni lo abbiamo visto con la funzione del linguaggio C `printf`, che ha un primo argomento che serve a determinare il tipo e il numero degli argomenti che seguono:

```
/** Dichiarazione, nel file stdio.h */
int printf(const char *format, ...) ;


/** Utilizzo */
printf("stringa: %s; intero: %d", "codice errore", -1);
```

Per indicare degli argomenti variabili, nella dichiarazione di una funzione, si utilizzano tre punti, dopo i parametri fissi:

```
void log(int livello, int n_parametri, ...);
```

Nella definizione della funzione, per gestire i parametri, è necessario creare un ciclo di lettura utilizzando tre marco definite nel file `stdarg.h`:

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

`va_start` inizializza la lista di variabili `ap` affinché possa ricevere gli argomenti variabili; il parametro `lastfix` specifica quale sia l'ultimo parametro fisso della funzione.

`va_arg` torna il successivo parametro nella lista `ap`; il parametro `type` indica il tipo di dato del parametro.

`va_end` termina l'elaborazione dei parametri e deve essere sempre chiamata prima che la funzione termini.

```
/**
 * @file funzioni-variabili.cpp
 * Esempio di funzione con parametri variabili.
 */

#include <iostream>
#include <iomanip>
#include <fstream>

#define LOG_DEBUG   1
#define LOG_AVVISO  2
#define LOG_ERRORE  3

#define S_DEBUG     "debug"
#define S_AVVISO    "avviso"
#define S_ERRORE    "errore"

#define ERR_NO_FILE_NAME  -1
#define ERR_NO_FILE_OPEN  -2

#define S_ERR_NO_FILE_NAME  "specificare il path del file"
#define S_ERR_NO_FILE_OPEN  "impossibile aprire il file:"

using namespace std;

/** Funzione di output con parametri variabili */
void log(int livello, int n_parametri, ...)
{

    /** Definisce la spaziatura del primo campo */
    cerr << setw(8);

    /** Scrive il livello del messaggio */
    switch(livello) {
        case LOG_DEBUG:  cerr << S_DEBUG ; break;
        case LOG_AVVISO: cerr << S_AVVISO; break;
        default:         cerr << S_ERRORE; break;
    }

    /** Scrive il carattere di separazione */
    cerr << " | ";
```

```
    /** Dichiara la variabile per la lista dei parametri */
    va_list lista_parametri;

    /**
     * Inizializza la lista dei parametri e considera
     * tutti gli argomenti dopo n_parametri come variabili.
     */
    va_start(lista_parametri, n_parametri);

    /** Legge tutti i parametri nella lista e li scrive a video */
    for(int p = 1; p <= n_parametri; p++) {
        cerr << va_arg(lista_parametri, char*) ;
    }

    /** Chiude la lista dei parametri */
    va_end(lista_parametri);

    /** Scrive il testo del messaggio */
    cerr << endl;

}

int main(int argc, char** argv)
{
    /** Crea una variabile per gestire il file di output */
    ofstream doc;

    /** Puntatore per il nome del file di output */
    const char* filename = NULL;

    /** Se manca il nome del file di output, errore */
    if(argc < 2) {
        log(LOG_ERRORE, 1, S_ERR_NO_FILE_NAME);
        return ERR_NO_FILE_NAME;
    }

    /** Legge il nome del file di output */
    filename = argv[1];

    /** Prova ad aprire il file di output */
    doc.open (argv[1]);

    /** Se c'è stato un errore, lo segnala ed esce */
    if(!doc.is_open()) {
        log(LOG_ERRORE, 2, S_ERR_NO_FILE_OPEN, filename);
```

```
        return ERR_NO_FILE_OPEN;
    }

    /** OK, il file è pronto */
    log(LOG_AVVISO, 2, "ho aperto il file: ", filename);

    /** Scrive sul file di output */
    doc << "Testo del documento.\n";
    log(LOG_DEBUG, 2, "ho scritto sul file: ", filename);

    /** Chiude il file di output */
    doc.close();
    log(LOG_AVVISO, 2,  "ho chiuso il file: ", filename);

    return 0;
}
```

L'output di questo codice è:

```
% g++ src/cpp/funzioni-variabili.cpp -o src/out/esempio

% src/out/esempio
  errore | specificare il path del file

% src/out/esempio /esempio.txt
  errore | impossibile aprire il file:/esempio.txt

% src/out/esempio src/out/esempio.txt
  avviso | ho aperto il file: src/out/esempio.txt
   debug | ho scritto sul file: src/out/esempio.txt
  avviso | ho chiuso il file: src/out/esempio.txt```
```

A questo punto, non mi resta che parlarti delle funzioni che richiamano sé stesse, ovvero, le *funzioni ricorsive*:

```
/**
 * @file /src/funzioni-ricorsiva.cpp
 * Esempio di funzione ricorsiva.
 */

#include <iostream>
#include <iomanip>

using namespace std;

/** Dichiarazione della funzione ricorsiva */
```

```
long fattoriale(int n);

/** Funzione main prima della definizione */
int main(int argc, char** argv)
{
    int n = 9;
    cout << n << "! = " << fattoriale(n) << endl;
    return 0;
}

/** Definizione della funzione ricorsiva */
long fattoriale(int n)
{
    long fatt = 1;
    if(n > 1) {
        fatt = n * fattoriale(n-1);
    }
    return fatt;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/funzioni-ricorsiva.cpp -o src/out/esempio
> src/out/esempio
 1 2 3 4 5 6 7 8 9
```

Attenzione, però: una funzione che richiama sé stessa, se non è scritta bene, può generare una ricorsione in[de]finita, come due specchi posti l'uno di fronte all'altro. Lo spazio prospettico all'interno degli specchi è inesauribile, ma la memoria dei computer, per quanto estesa, ha i suoi limiti e un numero eccessivo di ricorsioni potrebbe generare uno *stack overflow*. Per questo motivo, bisogna ricorrere alla ricorsione solo nei casi in cui è indispensabile, preferendole, quando possibile, le istruzioni iterative e applicando sempre dei meccanismi di controllo della profondità di ricorsione che prevengano un sovraccarico dello stack.
Un altro tipo di ricorsione di cui diffidare è quella con cui si cerca, alle volte, di dare credibilità a una convinzione priva di fondamenti reali utilizzando un'altra convinzione simile.

In un romanzo sulla vita dello spadaccino giapponese Myamoto Musashi c'è una frase che è la logica conseguenza del suggerimento di Jacopone da Todi:

> Non bisogna interferire nel funzionamento dell'Universo, ma prima è necessario capire quale sia, il funzionamento dell'Universo

Il problema dei romanzi è che ti dicono spesso cosa fare, ma non ti spiegano quasi mai come farlo, o perché.
Diffida dei maestri che sanno solo insegnare, perché tutto ciò che ti raccontano

l'hanno imparato dai libri. Un buon maestro deve eccellere in qualcosa, che sia la scherma, il tiro con l'arco, la carpenteria o la manutenzione delle motociclette. Può non essere *il* migliore, ma deve essere *fra i* migliori; solo così, saprai che le sue idee sono valide. Al contrario, i maestri che non sono mai usciti da una scuola, non hanno mai dovuto mettere le loro idee alla prova dei fatti. Ti insegnano ciò che pensano sia giusto, ma ciò che è giusto o vero in una scuola, non sempre è altrettanto vero o giusto nel mondo reale.
È facile essere dei santi, in un monastero, fra persone che hanno la tua stessa cultura e i tuoi stessi principii; il difficile è restare dei santi anche fuori. Il Buddha Shakyamuni predicò la Benevolenza Universale perché visse in India, duemilaseicento anni fa, ma se fosse rimasto imbottigliato nel traffico di una città, dopo una giornata di lavoro, forse le sue idee sarebbero state più simili a quelle di Nietzsche.
Per fortuna, Musashi — quello vero, il Samurai —, fu sempre molto chiaro sia su ciò che è giusto fare che sul modo di ottenerlo. Nel *Libro dei Cinque Anelli*, diede ai suoi discepoli nove regole di vita:

- Non pensare in maniera disonesta.
- La Via è nel costante allenamento.
- Pratica molte arti.
- Conosci la Via e i modi di tutti i mestieri.
- Distingui vantaggi e svantaggi di ogni cosa.
- Sviluppa una comprensione intuitiva delle cose.
- Percepisci anche ciò che non può essere visto con gli occhi.
- Presta attenzione anche alle cose più insignificanti.
- Non perdere tempo in attività inutili.

Nel *Dokkodo*, scritto una settimana prima della sua morte, fu ancora più specifico:

- Non agire in maniera contraria al tuo destino.
- Non ricercare una vita facile.
- Non avere pregiudizi od ostilità per qualcosa.
- Pensa a te stesso con leggerezza e in maniera profonda agli altri.
- Sii distaccato dal desiderio.
- Non avere rimpianti per ciò che hai fatto.
- Non essere geloso degli altri.
- Non avere attaccamento per alcuna cosa.
- Non portare rancore.
- Non pensare alla vita sentimentale.
- Non avere né preferenze né avversioni.
- Sii indifferente al luogo in cui vivi.
- Non ricercare il cibo per il suo gusto.
- Fà che le future generazioni non siano legate ad armi antiche.
- Evita le superstizioni e i taboo.
- Utilizza solo gli strumenti necessarii e non conservare ciò che è inutile.

- Sii preparato a morire.
- Quando sei vecchio, ciò che possiedi non ha molta importanza.
- Rispetta Buddha e gli Dei, ma non contare sul loro aiuto.
- Non abbandonare il tuo onore, anche se ciò significa abbandonare la vita.
- Non deviare mai dalla Via.

Queste regole, che hanno permesso a Musashi di arrivare alla venerabile età di sessant'anni, dopo essere sopravvissuto vittorioso ad altrettanti combattimenti con tutti i migliori spadaccini del suo tempo, possono aiutarti a capire quale sia il funzionamento dell'Universo e cosa fare per non perturbarlo.
La pratica delle arti, siano esse intellettuali o marziali, e la conoscenza dei mestieri, unite all'attenzione per tutto ciò che ti circonda, aumenteranno il tuo bagaglio di esperienza e ti permetteranno di distinguere i pro e i contro di ogni situazione. Questo ti libererà dal demone dell'invidia, perché imparerai che tutte le condizioni, anche quelle apparentemente idilliache, hanno dei lati negativi. Il passo successivo sarà affrancarsi dal desiderio e dall'attaccamento alle cose: così come la funzione `log` ha bisogno di sapere quali siano i parametri fissi e quali siano quelli variabili, tu dovrai imparare a distinguere i tuoi desiderii dalle tue necessità, per sfuggire all'influsso dell'Annosa Dicotomia.
La Via, con la "V" maiuscola è simile a una via con la "v" minuscola. Lungo la via, incontri dei cartelli stradali, che ti indicano la direzione in cui procedere o la velocità da tenere, e dei cartelloni pubblicitarii, che ti segnalano delle attrazioni nelle vicinanze e ti invitano a deviare dal tuo cammino per andarle a visitare. Allo stesso modo, lungo la Via, troverai delle necessità, che ti instraderanno verso la tua destinazione e dei desiderii, che ti dis-trarranno dal tuo percorso e ti at-trarranno verso destinazioni alternative. Se tu agirai in base alle necessità, saprai sempre che ti stai muovendo nella direzione giusta, anche quando sarai costretto a rallentare o a percorrere strade che non gradisci. Se invece agirai in base ai desiderii andrai di qua e di là, come "un asino privo di briglie" e quando alla fine tornerai sulla strada giusta, potresti non avere più il tempo per arrivare alla tua destinazione.

Ti ho parlato delle regole di Musashi non perché siano le uniche disponibili, ma perché sono estremamente personali. Attingono ai principii di altre discipline, come il Buddismo, il Bushido o il Tao, ma sono *something else*, come direbbe Eddie Cochran. Tu dovrai fare altrettanto: imparare tutto ciò che puoi, tanto dai buoni quanto dai cattivi maestri, e poi definire le tue regole di vita, che potranno essere uguali, simili o del tutto differenti da quelle che ti sono state insegnate.
La Via, così come la Verità, è una modella, che ciascuno di noi ritrae dal suo punto di vista, cercando di intuire il corpo che si cela dietro alle pieghe del drappeggio. Ritrarla nello stesso modo in cui l'ha fatto un altro sarebbe sbagliato, perché il tuo punto di vista non è uguale al suo, ma guardarla da più punti di vista può aiutarti a capire meglio la sua forma. Musashi dice di prestare

attenzione anche alle cose insignificanti, Nan-in e Tenno sviluppano il loro *Zen di ogni istante*, Wittgenstein nei suoi *Diari*, scrive:

> Solo una cosa, è necessaria: essere capace di osservare tutto ciò che ti accade. Concentrarsi! Dio mi aiuti!

È chiaro che stanno tutti dipingendo la stessa immagine, anche se ciascuno lo fa con il suo stile. È per questo motivo, che Musashi prescrive di conoscere la Via degli altri mestieri: perché c'è sempre qualcosa da imparare, da chi fa bene il suo lavoro. Questo, per esempio, è il parallelo che lui fa fra lo stratega e il carpentiere:

> Per edificare una casa è necessaria un'accurata scelta dei materiali. Per i pilastri esterni si sceglieranno dei tronchi diritti e senza nodi, mentre per quelli interni si possono usare dei tronchi diritti con qualche piccolo difetto. Per le soglie, gli architravi, gli infissi e le porte scorrevoli si useranno i legni migliori per l'aspetto, anche se non sono troppo robusti, e così via. Per le parti strutturali non è importante l'aspetto estetico quanto la robustezza. Il legname meno pregiato e con molti nodi viene invece utilizzato per i ponteggi e, alla fine, viene bruciato.

Ciò che è vero per il carpentiere, cambiando il punto di vista, è vero anche per lo stratega e potrà esserlo anche per te, se ti troverai ai gestire un progetto o un gruppo di lavoro. Anche come programmatore, comunque, dovrai tenerti costantemente aggiornato sulle tecniche e sui linguaggi di programmazione e dovrai conoscere il modo in cui lavora chi si occupa delle basi-dati o dei sistemi. Dovrai essere capace di identificare i malfunzionamenti anche se non hanno effetti sull'interfaccia grafica e potrai riuscirci solo se presterai attenzione anche al più piccolo dettaglio. Cambiano i nomi, perché sono passati quattro secoli, ma i problemi restano gli stessi, così come le soluzioni.
Del resto, arte, scienza, filosofia e religione sono tutti tentativi di dare una risposta alle stesse domande sulla nostra esistenza. Dato che l'oggetto di studio è lo stesso, è più che normale che, alle volte, le risposte si somiglino, allo stesso modo in cui l'algoritmo per il calcolo del fattoriale di un numero sarà più o meno lo stesso, indipendentemente dal linguaggio di programmazione.

Istruzioni condizionali

> Fare qualcosa d'intenzionale implica capire che esistono delle alternative, e sceglierne una; e questi sono attributi dell'intelligenza artificiale

Le istruzioni condizionali sono l'elemento più importante del codice.

Ogni programma deve saper reagire correttamente al variare delle condizioni di utilizzo; per far ciò, si utilizzano le cosiddette *istruzioni condizionali*, che permettono di definire il comportamento del sistema a seconda che una determinata condizione si riveli vera o falsa.
Il C++ possiede due tipi di istruzione condizionale: le sequenze `if-else` e l'istruzione `switch`.
La forma generale delle istruzioni `if-else` è la seguente:

```
if ( <condizione> ) {
    // istruzioni da eseguire se la condizione è vera
} else {
    // istruzioni da eseguire se la condizione è falsa
}
```

Se l'espressione condizionale all'inizio del codice è vera, il programma eseguirà il primo blocco di istruzioni; se no, eseguirà il secondo blocco di istruzioni.

```
if ( a > 8 ) {
    cout << "maggiore" << endl;
} else {
    cout << "minore" << endl;
}
```

Se la condizione falsa non richiede alcuna azione specifica, il secondo blocco di istruzioni può essere omesso:

```
typedef Importo unsigned long;

Importo raddoppiaStipendio(Importo stipendioCorrente)
{
    if ( stipendioCorrente > 0 ) {
        stipendioCorrente *= 2;
    }
    return stipendioCorrente;
}
```

Allo stesso modo, le parentesi graffe possono essere omesse se il blocco istruzioni che racchiudono è costituito da una singola istruzione:

```
if ( a > 8 )
    cout << "maggiore" << endl;
else
    cout << "minore" << endl;
```

Personalmente, trovo che questa forma sia inelegante e che renda il codice meno chiaro, favorendo quindi gli errori. La utilizzo solo nelle istruzioni di gestione degli errori, dove il flusso del programma si interrompe bruscamente, perché l'aspetto sgraziato dell'istruzione evidenzia l'eccezione, rendendo il codice più auto-esplicativo.

```
if ( divisore == 0 )
    throw std::runtime_error("errore");
```

Se le condizioni da valutare sono più di due, si possono concatenare più istruzioni condizionali utilizzando l'istruzione `else if`, che permette di definire una condizione alternativa alla prima e di associarle un blocco di codice. Anche in questo caso, si può chiudere la sequenza con un'istruzione `else`, definendo un blocco di istruzioni da eseguire se non si verifica nessuna delle condizioni previste.

```
if ( <prima condizione> ) {
    /*
     * istruzioni da eseguire se
     * la prima condizione è vera
    */
} else if ( <seconda condizione> ) {
    /*
     * istruzioni da eseguire se
     * la prima condizione è vera
     */
} else {
    /*
     * istruzioni da eseguire se nessuna
     * delle due condizioni è vera
     */
}
```

Le istruzioni `if-else` influenzano la leggibilità del codice; è una cosa di cui il buon programmatore deve sempre tenere conto. Il C++ è un linguaggio indipendente dalla formattazione, quindi, una stessa istruzione può essere scritta in molte maniere diverse:

```
if ( <condizione> )
{
    ...
}
else
{
    ...
}
```

o, pure:

```
if ( <condizione> ) {
    ...
} else {
    ...
}
```

o perfino:

```
if ( <condizione> ) { ...} else { ... }
```

Se le istruzioni sono poche e semplici, una forma vale l'altra (fatte salve le questioni di stile, ovviamente), ma se il flusso del programma fosse, come di solito avviene, più complesso, è necessario fare in modo che la forma dell'istruzione semplifichi la scrittura, la lettura e un'eventuale correzione del codice.
Immagina un brano di codice che debba fare una verifica all'inizio dell'elaborazione e, a seconda dell'esito, eseguire una sequenza di istruzioni o inviare un messaggio di errore:

```
if ( <condizione> ){
    /*
     * righe di codice da
     * eseguire in caso la
     * condizione sia vera
     */
} else {
    /* gestione dell'errore */
}
```

Se le istruzioni da eseguire in caso di buon successo della verifica sono poche e semplici, questa sequenza non darà problemi, ma se, al contrario, le istruzioni fossero tante e complesse, leggendo il codice potresti arrivare all'istruzione `else` e non ricordarti più a quale condizione fosse associata. In questi casi, io preferisco la forma:

```
if ( <errore> ) {
    /* gestione dell'errore */
} else {
    /*
     * righe di codice da
     * eseguire in caso la
     * condizione sia vera
     */
}
```

Dato che la gestione dell'errore non richiederà mai più di qualche riga di codice, potrai capire a colpo d'occhio tutto il flusso del programma, indipendentemente dalla lunghezza del secondo blocco di istruzioni.
Tutto questo, ovviamente, non vuole essere né un invito né una giustificazione per la scrittura di istruzioni complesse. A meno che non sia necessario limitare le chiamate a funzione per garantire un'alta velocità di esecuzione, è sempre meglio scomporre il flusso del programma in una serie di funzioni distinte e specializzate. Renderai il tuo programma un po' più lento (o, meglio: un po' meno veloce), ma il codice sarà molto più facile da leggere o da modificare.

Immagina adesso un brano di codice che richieda molte condizioni `if` concatenate:

```
esito = 0;

if ( <condizione 1> ) {
    esito = 1;
} else if ( <condizione 2> ) {
    esito = 2;
} else if ( <condizione 3> ) {
    esito = 3;
} else {
    esito = 9;
}

return esito
```

Questa forma, per quanto corretta e formalmente ineccepibile, potrebbe rivelarsi difficile da gestire se le condizioni da considerare fossero molto complesse o numerose. Il buon programmatore, allora, può decidere di contravvenire alla (giusta) norma che prescrive di non inserire delle istruzioni `return` all'interno del codice, e scrivere la sequenza in questo modo:

```
esito = 0;

if ( <condizione 1> ) {
    return 1;
}
if ( <condizione 2> ) {
    return 2;
}
if ( <condizione 3> ) {
    return 3;
}

return 9
```

Non ti sto dicendo che sia giusto scrivere così e vedi da solo che il codice è rozzo e inelegante, ma ci potrebbero essere dei casi in cui sia questa, la forma da preferire. Per esempio, per un sistema che generi del codice in maniera automatica, è molto più semplice gestire delle istruzioni `if` isolate che delle condizioni `if-else` concatenate. Pensa a una *stored-procedure* che debba controllare l'integrità referenziale dei parametri ricevuti:

```
CREATE FUNCTION utente_insert (
  _id_classe INTEGER
, _id_gruppo INTEGER
, _username  VARCHAR(255)
```

```
, _cognome   VARCHAR(80)
, _nome      VARCHAR(80)
)
RETURNS INTEGER DETERMINISTIC
BEGIN
    DECLARE _id    INTEGER DEFAULT -1;
    DECLARE _count INTEGER DEFAULT 0;

    SELECT count(*) INTO _count FROM classe WHERE (id = _id_classe);
    IF _count = 0 THEN
        RETURN -2;
    END IF;

    SELECT count(*) INTO _count FROM gruppo WHERE (id = _id_gruppo);
    IF _count = 0 THEN
        RETURN -3;
    END IF;

    IF (_username IS NULL) OR (_username = '') THEN
        RETURN -4;
    END IF;

    IF (_cognome IS NULL) OR (_cognome = '') THEN
        RETURN -5;
    END IF;

    IF (_nome IS NULL) OR (_nome = '') THEN
        RETURN -6;
    END IF;

    ...
```

Se scrivi il codice in questa maniera, puoi inserire o rimuovere un parametro (e i relativi controlli) senza alterare il resto del codice, cosa che non avverrebbe se tu concatenassi le istruzioni `if`. Perderai un po' di velocità di esecuzione, ma il codice sarà molto più facile da scrivere o da modificare.
Attento, però: mettere in sequenza delle semplici istruzioni `if` è cosa ben diversa dal creare una catena di istruzioni `else-if` perché, se in caso di errore non blocchi l'elaborazione con un'istruzione `return`, il programma andrà avanti verificando le condizioni seguenti e l'errore nella prima condizione potrebbe ripercuotersi sul codice successivo:

```
/** Qui comincia il male.. **/
if ( divisore == 0 ) {
    cout << "Errore: divisione per zero" << endl;
}
```

```
/** ..e il peggio lo segue **/
if ( (dividendo / divisore) > 1 ) {
    ...
}
```

Non avendo un'istruzione `return` il codice della prima verifica non bloccherà l'esecuzione della funzione, che andrà in errore quando proverà a eseguire una divisione per zero.

L'istruzione `switch` permette di gestire più casi, basandosi sulla valutazione di una espressione:

```
switch(<espressione>)
{
    case <costante> : istruzioni... [break];
    case <costante> : istruzioni... [break];
    ...
    default: istruzioni...
}
```

Le parole-chiave `case` e `default` identificano i valori gestiti dall'istruzione `switch`. I `case` possono (ed è utile che siano) più di uno, ma le costanti associate a ciascuno di essi devono avere dei valori diversi. La condizione `default`, al contrario, deve essere unica.
L'esecuzione dell'istruzione inizia al `case` la cui costante è uguale al valore dell'espressione di `switch` e termina alla parola chiave `break`. Se l'espressione ha un valore non previsto dai `case`, l'istruzione esegue il codice associato all'etichetta `default`:

```
#include <iostream>
#include <cstdlib>

#define POS_NESSUNO -1
#define POS_ERRORE   0
#define POS_MERCURIO 1
#define POS_VENERE   2
#define POS_TERRA    3
#define POS_MARTE    4
#define POS_GIOVE    5
#define POS_SATURNO  6
#define POS_URANO    7
#define POS_NETTUNO  8
#define POS_PLUTONE  9

using namespace std;

int main(int argc, char** argv)
```

```
{
    int pianeta;

    /**
     * Legge i parametri di input.
     * Se ce ne sono, prova a convertire il primo parametro
     * in un intero. Se il parametro non è un intero, la
     * funzione atoi torna 0.
     */
    if(argc > 1) {
        pianeta = atoi(argv[1]);
    } else {
        pianeta = POS_NESSUNO;
    }

    /** Gestisce i casi possibili */
    switch( pianeta ) {
        case POS_ERRORE:   cout << "Valore non valido";
            break;
        case POS_MERCURIO: cout << "Mercurio";
            break;
        case POS_VENERE:   cout << "Venere";
            break;
        case POS_TERRA:    cout << "Terra";
            break;
        case POS_MARTE:    cout << "Marte";
            break;
        case POS_GIOVE:    cout << "Giove";
            break;
        case POS_SATURNO:  cout << "Saturno";
            break;
        case POS_URANO:    cout << "Urano";
            break;
        case POS_NETTUNO:  cout << "Nettuno";
            break;
        case POS_PLUTONE:  cout << "Plutone";
            break;
        default:
            cout << "Inserire un valore da: "
                 << POS_MERCURIO
                 << " a "
                 << POS_PLUTONE;
    }

    cout << endl;
```

```
    return 0;
}
```

Compilando ed eseguendo questo codice, otterrai:

```
% g++ src/cpp/istruzioni-condizionali-switch.cpp -o src/out/esempio
% src/out/esempio
Inserire un valore da: 1 a 9
% src/out/esempio 4
Marte
% src/out/esempio terra
Valore non valido
```

In sostanza: con una `break` alla fine di ciascun caso, l'istruzione `switch` è una forma più elegante (ed efficiente) dell'istruzione `if - else if`:

```
if(piaeta == POS_ERRORE) {
   cout << "Valore non valido";
} else if(pianeta == POS_MERCURIO) {
    cout << "Mercurio";
} else if(pianeta == POS_VENERE) {
    cout << "Venere";
} else if ..
```

Se tu togliessi le interruzioni `break` alla fine di ciascun caso, l'output del programma sarebbe:

```
% src/out/esempio 4
MarteGioveSaturnoUranoNettunoPlutoneInserire un valore da: 1 a 9
```

che in questo caso non ha senso, ma che può essere la scelta adatta se due casi possibili vanno elaborati nella stessa maniera.
Un'ultima cosa: ricordati sempre che, per dichiarare delle variabili all'interno dei `case`, è necessario aggiungere delle parentesi graffe; altrimenti, avrai un errore in fase di compilazione:

```
switch( x ) {
    case 1:
        int y = 9;      /** errore di complilazione */
        cout << x + y;
        break;
    case 2: {
        int y = 2;      /** corretto */
        cout << x + y ;
        break;
    }
    default:
        cout << "default" << endl;
}
```

La vita ci chiede spesso di fare delle scelte condizionali. Quando ciò avviene, hai due possibilità: o fai la scelta più conveniente per te o fai la scelta che ti sembra più conveniente per il maggior numero di persone per il più lungo periodo di tempo possibile. Nel primo caso sarai un vettore di Entropia, mentre nel secondo caso sarai un paladino della Gravità.
Come sai, per il C'hi++ la scelta esatta (inteso come participio passato del verbo *esigere*) è la seconda: tutta la materia non è che la manifestazione di una unica Energia, quindi ha poco senso distinguerci gli uni dagli altri; dobbiamo invece ragionare come Sa^{c}di di Shirāz, quando dice:

> Son membra d'un corpo solo i figli di Adamo, da un'unica essenza
> quel giorno creati. E se uno tra essi a sventura conduca il destino,
> per le altre membra non resterà riparo.

Cercare il proprio tornaconto personale a discapito degli altri è sbagliato. Bisogna comportarsi bene e cercare di convincere anche gli altri a fare altrettanto, perché, come recita il *Mantiq al-Tayr*:

> tutto il male o il bene che feci, in verità lo feci solo a me stesso.

Ma come si fa a capire cosa sia *bene*? Ci sono casi in cui è facile dare la scelta giusta, come nel caso del maestro Zen Bokuju:

```
switch( stimolo ) {
    case fame:
        azione = mangio;
        break;
    case sonno:
        azione = dormo;
        break;
    case sete:
        azione = bevo;
        break;
}
```

ma altre volte ci troviamo di fronte a scelte più complesse:

> Una ragazza è rimasta incinta a séguito di una violenza: può decidere
> di abortire?

oppure:

> Un uomo, condannato per omicidio, in carcere ha ucciso altri due
> carcerati e una guardia: va condannato a morte o no?

Se queste domande le fai a un cattolico, lui — coerentemente con il suo Credo — ti risponderà che no, non è possibile né abortire né condannare a morte perché la vita è un dono di Dio e nessuno ce ne può privare. Se invece queste domande le poni a un Giudice, avrai risposte diverse a seconda della Nazione

a cui appartiene, perché mentre sottrarre dei beni materiali è considerato un reato ovunque, esistono degli Stati in cui è permesso sottrarre a un individuo il bene più prezioso che ha.
Un tempo, i credenti mettevano al rogo gli scienziati, accusandoli di eresia; il 6 Giugno del 1945, però, la Scienza ha mostrato al Mondo il suo potere ed è diventata di fatto il nuovo Dio per milioni di persone; da allora, le parti si sono invertite e adesso sono gli scienziati a mettere al rogo ogni forma di spiritualità. Il problema è che se privi la giurisprudenza di una base spirituale, quello che otterrai sono Leggi *pret-a-porter*, rimedii temporanei a delle esigenze contingenti. Nella migliore delle ipotesi.
La teocrazia è un errore, ma anche la *a*-teocrazia dev'essere evitata. La Fede è stata la colla che ha tenuto unita la nostra società per quasi duemila anni. Forse quella colla era solo una nostra invenzione, ma lo sono anche gli Stati, il denaro, i Diritti Umani, le Leggi. Nessuno di questi concetti così importanti per la nostra Società esiste davvero, ma li utilizziamo lo stesso perché, come il linguaggio *C*, pur essendo solo delle convenzioni, sono utili al loro scòpo.
Ora che questa colla non c'è più, le scelte dei legislatori non sono più mosse dal perseguimento di un obiettivo comune (corretto o sbagliato che fosse), ma dalla ricerca dell'approvazione di un elettorato composto in buona parte da zombie culturali e da egoisti che perseguono unicamente il proprio interesse momentaneo: il pascolo ideale per demagoghi con aspirazioni dittatoriali. La minoranza di idealisti e di persone colte, priva di valori trascendenti, non può che agire in base ai propri sentimenti o ai propri auspici e subisce inevitabilmente il malefico influsso dell'Annosa Dicotomia: fanno scelte che puntano al bene comune, ma si tratta di un bene comune molto spesso miope e temporaneo. Come scacchisti mediocri, vedono ciò che è bene qui e ora, ma non riescono a valutarne le conseguenze a lungo termine.
Pensa a quale potrebbe essere, secondo te, la soluzione giusta alle due domande che ti ho fatto e poi pensa al *motivo* quella soluzione ti appare giusta. Perché permettiamo la soppressione di un feto che non ha fatto del male a nessuno, mentre lasciamo in vita chi ne ha già fatto? Vogliamo fare la cosa giusta o vogliamo solo sentirci buoni?
Il Maestro Canaro si fece molti nemici con la sue idee sull'aborto. Anche alcune persone che inizialmente lo avevano appoggiato lo accusarono di cercare l'appoggio della Chiesa Cattolica, mentre stava solo applicando il precetto del *Metta Sutra* che predica la felicità non solo per tutti coloro che sono nati, ma anche per coloro che devono nascere:

> bhåtà và sambhavesã và sabbe sattà bhavantu sukhitattà

bhåtà quelli che sono nati
và o
sambhavesã quelli che cercano la nascita
và o
sabbe tutti
sattà gli esseri
bhavantu possano essere

sukhitattà felici nel loro cuore

Tutto questo a lui non importava: quando gliene parlai, mi disse che preferiva perdere un milione di seguaci che una vita.

Rispondere alla domanda sul condannato è più difficile. Lo scòpo delle tue azioni deve essere, come sempre, il miglioramento: chi sbaglia deve capire che ha fatto un errore e non ripeterlo in altri cicli di esistenza:

```
int pentimento( azione ){

    bool ripetere;

    if ( azione == errore ) {
        ripetere = false;
    } else {
        ripetere = true;
    }

    return ripetere;
}
```

Se il peccatore è davvero pentito, allora è giusto che sia assolto, perché, come dice Attar:

> Cento Mondi di peccato sono dissipati dalla luce di un solo pentimento.

Ma il pentimento dev'essere reale: il peccatore deve detestare il suo errore e scegliere di morire piuttosto che ripeterlo ancora.
Tagliare una mano a chi ruba, costringendolo a portare il cibo alla bocca con la stessa mano con cui si pulisce il sedere, è un metodo un po' drastico, ma efficace per costringere qualcuno a meditare sull'insensatezza delle sue azioni passate — specie in un luogo come il deserto, dove i bidet sono più rari che in Francia. La *Lex Talionis* può funzionare per reati minori, perché chi la subisce ha il tempo di riflettere sui suoi errori, ma nel caso di un omicidio non solo è contraria all'obbligo di benevolenza che abbiamo nei confronti degli altri esseri senzienti, ma potrebbe anche essere controproducente, perché se il condannato non capisce il suo errore prima di morire è possibile che le sue azioni delittuose vengano ripetute in altri cicli dell'Universo.
D'altro canto, abbiamo un obbligo di benevolenza anche nei confronti degli altri carcerati e delle guardie carcerarie, quindi non possiamo lasciare che il condannato li uccida. La soluzione ideale sarebbe quella di metterlo in condizione di non nuocere a terzi, lasciandolo poi meditare sui suoi errori, ma se questo non fosse possibile, come ci dovremmo comportare? Se un individuo ripete più volte lo stesso atto delittuoso evidentemente non capisce o non vuole capire il suo errore. Se non capisce non è *senziente*, nel senso di *sensibilità*, quindi non

può concorrere al miglioramento dell'Universo. Se non è utile al miglioramento, possiamo considerarlo alla stessa stregua del gatto di Nansen:

> Nansen un giorno vide i monaci delle sale Orientali e Occidentali che litigavano per un gatto. Egli sollevò il gatto e disse: "Se mi direte una parola di Zen, salverò il gatto; se no, lo ucciderò".
> Nessuno seppe rispondere e Nansen tagliò il gatto in due.

La morte del condannato, però, se mai dovesse rendersi necessaria, non deve essere considerata una vendetta di cui gioire, ma un evento tanto doloroso quanto inevitabile, di cui dolersi come di un'amputazione. Ciascuno, in quel giorno, dovrebbe chiedersi se, con *pensieri, parole, opere e omissioni*, non abbia contribuito in qualche modo a quella perdita. Una Società che esalta l'individualismo, il successo e il denaro non può dirsi del tutto innocente se chi non ha i mezzi o la capacità di ottenerli in maniera lecita cerca di procurarseli in altro modo.

Un insegnante buddhista, saputo che il Maestro Canaro, nei suoi scritti, sosteneva che non c'è modo di sottrarsi al ciclo delle rinascite, si recò da lui e, deciso a dimostrare che si sbagliava, lo sfidò a un *Dharma Combat* per chiarire le reciproche posizioni. Il Maestro Canaro rispose che non sapeva cosa fosse un *Dharma Combat*; al che, l'insegnante buddhista spiegò che era un confronto dialettico, per dimostrare la propria conoscenza della dottrina. Il Maestro Canaro allora annuì e disse: "Va bene, ma prima che cominciamo, dimmi se tu, questo confronto, lo vuoi vincere o perdere, in modo che io possa accontentarti." Sentendo quelle parole, l'insegnante buddhista si rese conto che le sue intenzioni non erano pure: non voleva quella sfida per arrivare alla verità, ma solo per il piacere della vittoria. Così si inchinò, ringraziò il Maestro Canaro per avergli fatto capire quella sua debolezza e, da quel momento in poi, divenne un suo discepolo.

Istruzioni iterative

> Invero è cosa piuttosto strana a udirsi, ma il nome che noi almeno vi diamo – un nome a cui, per difetto di esperienza in materia, non penserebbe nessuno – è: Spazionismo

Le istruzioni di ciclo sono le componenti fondamentali della programmazione.

Ci sono tre tipi di istruzioni di ciclo:

- `for`
- `while`
- `do while`.

Queste istruzioni sono composte di due parti: un'istruzione di *controllo del ciclo*, che ne determina la durata e un *corpo del ciclo*, composto dalle istruzioni che vengono ripetute ad ogni iterazione. La ripetizione può protrarsi o per un determinato numero di volte o fino a che non sia raggiunta una determinata condizione.

Il ciclo `for` viene utilizzato quando vogliamo eseguire il ciclo in numero determinato di volte. La forma generale è la seguente:

```
// controllo del ciclo
for(<stato iniziale> ; <stato finale> ; <variazione>)
{
    // corpo del ciclo
    <istruzioni>
}
```

Le tre condizioni all'interno delle parentesi sono utilizzate dall'istruzione `for` per controllare l'esecuzione delle istruzioni all'interno del corpo del ciclo.
La prima espressione è valutata solo una volta all'inizio del ciclo e, solitamente, serve a inizializzare le variabili utilizzate.
La seconda espressione è una condizione logica o relazionale che viene valutata all'inizio di ogni iterazione: se torna `0` o `false` l'esecuzione del ciclo termina, altrimenti prosegue.
La terza espressione viene valutata al termine di ogni iterazione e, di solito, è costituita da un'espressione di incremento o decremento delle variabili utilizzate per il controllo del ciclo. Per fare ciò, si utilizzano degli operatori unarii (ovvero che operano su una singola variabile) detti: *operatori di incremento* e *operatori di decremento*, che possono svolgere la loro funzione o prima o dopo l'utilizzo della variabile, a seconda che vengano posti prima o dopo l'identificatore della variabile:

```
x++     // valuta x e poi la incrementa di un'unità
++x     // incrementa x di un'unità e poi la valuta
x--     // valuta x e poi la riduce di un'unità
--x     // riduce x di un'unità e poi la valuta
```

Se `x` è una variabile di tipo intero o in virgola mobile l'incremento è di un'unità aritmetica; se invece `x` è un puntatore l'incremento equivale alla dimensione della variabile a cui il puntatore riferisce, come ti ho fatto vedere parlando dei tipi di dato.
In questo caso, la variabile `p` è un intero, quindi l'istruzione:

```
for ( p = POS_MERCURIO; p <= POS_PLUTONE; p++ ) {
```

incrementerà la variabile `p` di 1.

```
#include <iostream>
#include <cstdlib>

#define POS_NESSUNO -1
```

```
#define POS_ERRORE   0
#define POS_MERCURIO 1
#define POS_VENERE   2
#define POS_TERRA    3
#define POS_MARTE    4
#define POS_GIOVE    5
#define POS_SATURNO  6
#define POS_URANO    7
#define POS_NETTUNO  8
#define POS_PLUTONE  9

using namespace std;

/**
 *  mostraPianeta
 *  Visualizza il nome di un pianeta, data la sua posizione.
 *  @param  int  pianeta Posizione del pianeta.
 *  @return bool esiste  true se il pianeta esiste.
 */
bool mostraPianeta(int pianeta )
{
    bool esiste = true;

    switch( pianeta ) {
        case POS_ERRORE:
            cout << "Valore non valido";
            break;
        case POS_MERCURIO:
            cout << "Mercurio";
            break;
        case POS_VENERE:
            cout << "Venere";
            break;
        case POS_TERRA:
            cout << "Terra";
            break;
        case POS_MARTE:
            cout << "Marte";
            break;
        case POS_GIOVE:
            cout << "Giove";
            break;
        case POS_SATURNO:
            cout << "Saturno";
            break;
        case POS_URANO:
```

```
            cout << "Urano";
            break;
        case POS_NETTUNO:
            cout << "Nettuno";
            break;
        case POS_PLUTONE:
            cout << "Plutone";
            break;
        default:
            esiste = false;
            cout << "Inserire un valore da: "
                 << POS_MERCURIO
                 << " a "
                 << POS_PLUTONE;
    }

    return esiste;
}

/**
 *  main
 *  Funzione principale del programma, richiama la funzione
 *  mostraPianeti passandole i valori da 1 a 9.
 */
int main(int argc, char** argv)
{
    int p = POS_NESSUNO;

    /** Elenca tutti i Pianeti del Sistema Solare */
    for ( p = POS_MERCURIO; p <= POS_PLUTONE; p++ ) {
        cout << p << ": ";
        mostraPianeta( p );
        cout << endl;
    }

    return 0;
}
```

La prima istruzione inizializza il valore di `p` a `1` (il valore della costante `POS_MERCURIO`) e prosegue, per incrementi successivi di 1, fino a che il valore di `p`è minore o uguale a `9` (il valore della costante `POS_PLUTONE`). A ogni ripetizione il programma mostra il valore della variabile `p`, richiama la funzione `mostraPianeta`, passandole il valore corrente di `p`, poi aggiunge un a capo.
L'output di questo programma è:

```
% g++ src/cpp/istruzioni-iterative-for.cpp -o src/out/esempio
% src/out/esempio
```

```
1: Mercurio
2: Venere
3: Terra
4: Marte
5: Giove
6: Saturno
7: Urano
8: Nettuno
9: Plutone
```

Si può utilizzare un ciclo `for` anche per effettuare cicli con un numero indefinito di iterazioni, basta omettere le tre espressioni di controllo, mantenendo solo delle *istruzioni nulle*, composte dal solo terminatore `;`:

```
for( ; ; )
{
    // istruzioni
}
```

Un ciclo di questo tipo continuerà a ripetersi indefinitamente e, se non viene fermato in qualche maniera, causerà inevitabilmente dei problemi al computer che lo esegue. È necessario quindi porre un limite al numero di ripetizioni, utilizzando la stessa parola-chiave `break` che abbiamo usato con le istruzioni `switch`. Stavolta, però, cominceremo a fare le cose come vanno fatte e separeremo le tre componenti del programma precedente in tre file distinti: `pianeti.h`, che conterrà le dichiarazioni delle costanti e della funzione `mostraPianeta`; `pianeti.cpp`, contenente la definizione della funzione `mostraPianeta` e `pianeti-main.cpp` per la funzione `main`:

```
/**
 * @file pianeti.h
 * Costanti e funzione per la gestione dei Pianeti.
 */

#ifndef _PIANETI
#define _PIANETI 1

using namespace std;

/** Dichiarazione delle costanti */
#define POS_NESSUNO -1
#define POS_ERRORE   0
#define POS_MERCURIO 1
#define POS_VENERE   2
#define POS_TERRA    3
#define POS_MARTE    4
#define POS_GIOVE    5
#define POS_SATURNO  6
```

```
#define POS_URANO    7
#define POS_NETTUNO  8
#define POS_PLUTONE  9

/** Dichiarazione delle funzioni */
bool mostraPianeta(int pianeta);
string nomePianeta(int posizione);

#endif /* _PIANETI */
```

```
/**
 * @file pianeti.cpp
 * @version 1.0
 * Funzione per la gestione dei Pianeti.
 */

#include <iostream>
#include "pianeti.h"

/**
 *  mostraPianeta
 *  Visualizza il nome di un pianeta, data la sua posizione.
 *  @param  int  pianeta Posizione del pianeta.
 *  @return bool esiste  true se il pianeta esiste.
 */
bool mostraPianeta(int pianeta)
{
    bool esiste = true;

    switch( pianeta ) {
        case POS_ERRORE:
            cout << "Valore non valido";
            break;
        case POS_MERCURIO:
            cout << "Mercurio";
            break;
        case POS_VENERE:
            cout << "Venere";
            break;
        case POS_TERRA:
            cout << "Terra";
            break;
        case POS_MARTE:
            cout << "Marte";
            break;
        case POS_GIOVE:
```

```
            cout << "Giove";
            break;
        case POS_SATURNO:
            cout << "Saturno";
            break;
        case POS_URANO:
            cout << "Urano";
            break;
        case POS_NETTUNO:
            cout << "Nettuno";
            break;
        case POS_PLUTONE:
            cout << "Plutone";
            break;
        default:
            esiste = false;
            cout << "Inserire un valore da: "
                 << POS_MERCURIO
                 << " a "
                 << POS_PLUTONE;
    }

    return esiste;
}
```

```
/**
 * @file pianeti-main.cpp
 * Funzione principale per il programma di gestione dei Pianeti.
 */

#include <iostream>
#include "pianeti.h"

using namespace std;

int main(int argc, char** argv)
{
    int p = POS_MERCURIO;

    /** Elenca tutti i Pianeti del Sistema Solare */
    for ( ; ; ) {
        cout << p << ": ";
        /** Se incontra un errore, si ferma */
        if(!mostraPianeta( ++p )) break;
        cout << endl;
    }
```

```
    cout << endl;

    return 0;
}
```

L'incremento della variabile `p`, in questo caso, avviene all'interno dell'istruzione:

```
if(!mostraPianeta( p++ )) break;
```

La parola-chiave `break` è una delle tre *istruzioni di interruzione* che il C++ ha ereditato dal C; le altre due sono l'istruzione `continue`, che riporta l'elaborazione all'inizio del ciclo e l'istruzione `return`, che termina la funzione, restituendo un eventuale valore di ritorno alla funzione chiamante.

Per generare il programma, stavolta, dovremo passare al compilatore entrambi i file *.cpp*:

```
% g++ src/cpp/pianeti-main.cpp \
      src/cpp/pianeti.cpp \
   -o src/out/esempio
% src/out/esempio
1: Mercurio
2: Venere
3: Terra
4: Marte
5: Giove
6: Saturno
7: Urano
8: Nettuno
9: Plutone
10: Inserire un valore da: 1 a 9
```

Se invece avessimo scritto:

```
if(!mostraPianeta( ++p )) break;
```

il valore di `p` in tutte le istruzioni sarebbe stato maggiore di uno rispetto al valore corretto e il programma avrebbe "saltato" Mercurio perché il valore della variabile `p` sarebbe stato incrementato prima del suo utilizzo da parte della funzione:

```
% src/out/esempio
1: Venere
2: Terra
3: Marte
4: Giove
5: Saturno
6: Urano
7: Nettuno
```

```
8: Plutone
9: Inserire un valore da: 1 a 9
```

Si può utilizzare un ciclo `for` in questo modo, ma non ha molto senso. Meglio, invece, utilizzare l'istruzione di flusso `while`:

```
/**
 * @file pianeti-while-bool.cpp
 * Gestione dei Pianeti con istruzione while.
 */

#include <iostream>
#include "pianeti.h"

using namespace std;

int main(int argc, char** argv)
{
    int  p  = POS_MERCURIO;

    /** Assegna un valore alla variabile di controllo */
    bool ok = true;

    /** Ripete l'azione finché la variabile è uguale a true */
    while (ok) {

        /** Mostra il valore di p e lo incrementa */
        cout << p++ << ": ";

        /**
         * Mostra il nome sel pianeta e salva l'esito della
         * funzione nella variabile ok
         */
        ok = mostraPianeta(p);

        /** Aggiunge il solito a capo */
        cout << endl;
    }

    return 0;
}
```

L'output sarà uguale a quello della funzione che utilizzava il ciclo `for`:

```
% g++ src/cpp/pianeti-while-bool.cpp \
      src/cpp/pianeti.cpp \
   -o src/out/esempio
% src/out/esempio
```

```
1: Venere
2: Terra
3: Marte
4: Giove
5: Saturno
6: Urano
7: Nettuno
8: Plutone
9: Inserire un valore da: 1 a 9
```

Il ciclo `do-while` è uguale al ciclo `while` con la sola differenza che la condizione `while` viene valutata alla fine dell'iterazione e quindi il corpo del ciclo viene eseguito almeno per una volta. La forma generale del ciclo `do-while` è:

```
do
{
    <istruzioni>        // corpo del ciclo
} while(espressione)    // espressione di controllo
```

Non offenderò la tua intelligenza con un esempio; vorrei piuttosto farti notare una cosa grave: il codice di questi programmi è sgraziato. Il problema è che la funzione mostraPianeta fa troppe cose: non solo stabilisce il nome del Pianeta, ma lo stampa anche a video. In conseguenza di ciò, nel nostro output abbiamo anche quella brutta stringa di errore relativa alla posizione numero nove. Nel primo esempio in cui l'abbiamo utilizzata, questo non era un problema, ma adesso che il nostro programma si sta sviluppando, dobbiamo rendere ciascuna funzione più specialistica, dividendo l'elaborazione dei dati (capire quale sia il pianeta) dall'interfaccia utente (la stampa a video del nome). Per fare ciò, utilizzeremo una nuova funzione che aggiungeremo al file `pianeti.cpp`:

```
/**
 *  nomePianeta
 *  Torna il nome di un pianeta, data la sua posizione.
 *  @param  int    posizione Posizione del pianeta.
 *  @return string nome      Nome del pianeta o null, se errore.
 */
string nomePianeta(int posizione)
{
    string nome;

    /**
     * Questa funzione gestisce solo i casi esistenti
     * e lascia che sia la funzione chiamate a gestire
     * eventuali errori.
     */
    switch( posizione ) {
```

```
        case POS_MERCURIO:
            nome = "Mercurio";
            break;
        case POS_VENERE:
            nome = "Venere";
            break;
        case POS_TERRA:
            nome = "Terra";
            break;
        case POS_MARTE:
            nome = "Marte";
            break;
        case POS_GIOVE:
            nome = "Giove";
            break;
        case POS_SATURNO:
            nome = "Saturno";
            break;
        case POS_URANO:
            nome = "Urano";
            break;
        case POS_NETTUNO:
            nome = "Nettuno";
            break;
        case POS_PLUTONE:
            nome = "Plutone";
            break;
    }

    return nome;
}
```

La nuova funzione `main` sarà:

```
/**
 * @file pianeti-while-string.cpp
 * Gestione dei Pianeti con valori stringa.
 */

#include <iostream>
#include "pianeti.h"

using namespace std;

int main(int argc, char** argv)
{
    int    p  = POS_MERCURIO;
```

```
    string nome;

    /**
     * Scinde l'elaborazione del dato dalla sua
     * eventuale visualizzazione
     */
    while (!(nome = nomePianeta(p)).empty()) {
        cout << p++ << ": " << nome << endl;
    }

    return 0;
}
```

Il codice è sintatticamente più complesso, ma una volta capito che l'istruzione:

```
while(!(nome = nomePianeta(p)).empty())
```

significa:

> esegui la funzione `nomePianeta` passandole come parametro la variabile `p` finché non ti torna una stringa vuota

il flusso del programma diventa più evidente di quanto fosse nei casi precedenti. Il corpo del ciclo è passato da tre istruzioni a una e le due operazioni di elaborazione e visualizzazione sono ben distinte nel tempo. Oltre ad aver ottenuto un codice più facile da leggere, da correggere da eseguire e da modificare, ci siamo anche sbarazzati dell'odioso messaggio di errore. Direi che ne valeva la pena, no?

La religione dovrebbe aiutare l'Uomo a vivere meglio. Dovrebbe dare uno scòpo alla nostra esistenza, aiutarci a superare i momenti di dolore e definire una scala di valori che ci permetta di prendere delle decisioni in quei casi in cui il raziocinio o il semplice buon senso non possono essere d'aiuto. Finora, però, le religioni non hanno aiutato l'Umanità a vivere meglio, anzi: hanno avuto spesso l'effetto opposto perché sono state prese a pretesto per guerre, soprusi e contrasti più o meno violenti. Ciò dipende da due fattori: la natura umana e la mancanza di solidità logica dei loro principii. Infatti, dovendo interessarsi di argomenti che non possono essere sottoposti a un'analisi razionale, le religioni sono costrette a dedurre le regole della propria dottrina da una serie di dogmi non dimostrabili che i seguaci della religione — i quali, non a caso, sono detti: "fedeli" o: "credenti" — devono accettare per buoni senza metterli in discussione.
I dogmi sono i pilastri su cui si regge l'edificio della dottrina; se uno di essi si indebolisse o, peggio, se fosse rimosso, l'edificio rischierebbe di crollare, quindi ogni forma di eresia è vista dagli apparati ecclesiastici come un potenziale pericolo che va scongiurato con ogni mezzo, anche a costo di abiurare quegli stessi principii che si cerca di difendere. Questo, però, non fa che peggiorare le cose, perché i dogmi non sono leggi comprovabili, ma opinioni o speranze e ogni

tentativo di renderli più robusti ottiene l'effetto opposto perché si ampliano le dimensioni di una struttura che poggia su basi instabili.
Aristotele disse che:

> le scienze che derivano da un numero minore di premesse sono più rigorose delle scienze che ne discendono per mezzo dell'aggiunta di nuove premesse

In quest'ottica, il C'hi++ è una metafisica abbastanza rigorosa, perché richiede l'accettazione di due sole affermazioni non comprovabili e, di queste, solo una è strettamente necessaria alla coerenza interna della dottrina, l'altra è solo un auspicio.
Ai fedeli del C'hi++ è richiesto di credere, anche in assenza di prove o in presenza di prove contrarie (gli scienziati non sono infallibili: sono gli stessi che avevano visto dei canali su Marte) che ci sarà un momento in cui l'espansione dell'Universo terminerà e che tutto ciò che esiste tornerà a riunirsi nell'Uno primigenio:

```
void eternita()
{
    /**
     * La condizione è sempre vera, quindi il ciclo si ripete
     * indefinitamente
     */
    while(1) {

        unoPrimigenio();

        bigBang();

        espansione();

        contrazione();

        bigCrunch();

        /**
         * Il fatto che questa istruzione sia a commento è
         * una delle differenze fra C'hi++ e Buddismo.
         * if(satori()) break;
         */
    }
}
```

Se non ci fosse questa ciclicità, ovvero senza un'alternanza fra Entropia e Gravità, fra *Prakṛti* e *Puruṣa*, lo Spazionismo e, di conseguenza, il C'hi++ non avrebbero più senso, così come la nostra esistenza. La Vita si rivelerebbe un epifenomeno destinato a esaurirsi nella morte termica dell'Universo e nessun

ordine sociale sarebbe più possibile, perché ciascuno cercherebbe di ottenere il massimo possibile dai pochi anni che gli sono concessi, indifferente al costo che questo avrebbe per gli altri. Qualcosa di simile al *Black Friday* in un centro commerciale americano, per intendersi..

```
typedef struct {
    string evento;
    bool   esito;
} PostIt ;
```

Il secondo dogma del C'hi++ è l'esistenza di una memoria persistente dell'Universo che mantiene traccia dell'esito delle scelte fatte in ciascun ciclo; qualcosa di simile all'inconscio collettivo di Jung o ai *vāsanā* dell'Induismo:

> Ci sono due categorie di saṁskāra; la prima consiste nelle vāsanā, che sono impressioni lasciate nella mente dagli avvenimenti passati, tracce qui conservate allo stato latente ma pronte a manifestarsi in presenza delle condizioni adatte, cioè di situazioni analoghe a quelle che le hanno generate, e che le attiverebbero a causa della loro affinità. Sulla spinta delle vāsanā, una volta che siano attivate, e degli stati d'animo che queste manifestano, l'individuo presenta una tendenza inconscia ad agire in un determinato modo, e più in generale ad avere un certo tipo di comportamento, di sensibilità, di carattere; si tratta di una predisposizione innata che lo induce, nel bene come nel male, ad un comportamento analogo a quello che ha tenuto in passato, creando un circolo vizioso (o virtuoso) che si autoalimenta.

I *Post-It*, come li chiamava il Maestro Canaro, non sono indispensabili per il C'hi++, sono solo un'auspicio, perché la loro esistenza dà un senso alla nostra vita. Ho detto: *dà* in vece di: *darebbe* perché, così come avviene per gli esopianeti, noi non li "vediamo", ma possiamo inferire la loro esistenza dall'effetto che hanno su ciò che li circonda: istinto, premonizioni, *deja-vu*.
L'esistenza dei *Post-It* rende l'Universo *stateful* e quindi più appetibile per noi, ma anche in un Universo *stateless*, in cui ciascun ciclo di esistenza fa storia a sé, noi non avremmo né motivo né convenienza a comportarci in maniera egoistica. In primo luogo perché, come abbiamo già detto, essendo tutti la manifestazione di una stessa Energia, ciò che facciamo agli altri lo facciamo in realtà a noi stessi; in secondo luogo perché non è detto che in ciascun ciclo di esistenza il nostro io cosciente si manifesti nella stessa persona e quindi, se in un ciclo siamo *Jack the Ripper*, in un altro potremmo essere Mary Ann Nichols.
I *Post-It* furono sempre una spina nel fianco, per il Maestro Canaro, che spese gli ultimi anni della sua vita cercando un modo per ricondurli a qualcosa di reale o di eliminarli dalla dottrina, ma non riuscì a fare nessuna delle due cose. Inizialmente pensò la cosa più ovvia, ovvero che i Post-It fossero nel cervello, che fossero una particolare mappatura delle sinapsi contenente le istruzioni per reagire a determinate condizioni future. Ipotizzò anche che i sogni avessero la duplice funzione di prepararci a queste condizioni, anticipandocele e tenendo vi-

ve le connessioni cerebrali che avremmo dovuto sfruttare per affrontarle. Tutto questo, però, poteva essere solo una proiezione dei Post-It, una copia in memoria RAM di informazioni preservate altrove, perché, per poter essere persistenti, i Post-It devono trovarsi al di fuori dello *scope* della nostra esistenza, nella ROM o nell'*hard-disk* dell'Universo:

```
/** Dichiarazione della struttura PostIt */
typedef struct {
    string evento;
    bool   esito;
} PostIt ;

void eternita()
{
    /** Array di puntatori a PostIt */
    PostIt** vasana;

    while(1) {

        /** Inizio di un nuovo ciclo */
        unoPrimigenio();
        bigBang();

        /**
         *  L'array di eventi viene passato alle fasi
         *  espansione e di contrazione, che lo incrementano
         *  in funzione delle esperienze fatte dagli
         *  esseri senzienti
         */
        espansione(vasana);
        contrazione(vasana);

        /**
         *  Qui gli esseri senzienti sono annichiliti,
         *  ma l'informazione non si perde, perché è
         *  salvata altrove
         */
        bigCrunch();
    }
}
```

I Post-It, quindi, sono per definizione metafisici (o, nel caso del codice qui sopra, *meta-ciclici*) perché tutto ciò che è fisico verrà annichilito al termine di un ciclo di esistenza e rigenerato all'inizio del seguente. Possiamo credere nella loro esistenza, ma dobbiamo farlo per fede, in maniera dogmatica.
Il Maestro Canaro rifuggiva i dogmi; diceva che se tu imponi una verità, poi sei legato a essa e non puoi più cambiare idea, anche se ti accorgi di avere

sbagliato. Prendi il tuo libro, per esempio; quel sottotitolo che hai scelto: *Lo scopo della vita è il debug* ti impone di credere nei Post-It. Se il giorno dopo che l'hai pubblicato si scoprisse che non esistono, saresti costretto a ristamparlo o ad accettare il fatto che dice una cosa non vera. Il Maestro Canaro non voleva che questo avvenisse al C'hi++; per questo motivo, stabilì che dovesse avere un versionamento, come il software: perché potesse evolversi.

I letterati, gli scultori, i pittori non possono modificare le loro opere, una volta che sono state pubblicate. Possono dare un ritocco di colore qui, un colpo di scalpello là, ma si tratta sempre di aggiustamenti minimi, che non cambiano la struttura stessa dell'arte-fatto. I musicisti, i teatranti e, in parte, i cineasti sono un po' più fortunati, perché possono apportare più facilmente delle modifiche alle loro opere, ma si tratta comunque di eventi che accadono di rado.
Al contrario, la buona produzione di software ha il vantaggio di essere in continua evoluzione. Un software può essere *stabile*, ovvero non avere difetti noti, ma non è mai finito, completo, *perfetto*; sia perché l'utilizzo potrebbe rivelare dei difetti sfuggiti alla fase di test, sia perché delle variazioni del contesto di utilizzo potrebbero richiedere delle modifiche al sistema. Il buon software viene quindi costantemente aggiornato e le diverse versioni di uno stesso prodotto sono numerate in maniera progressiva con dei codici composti da tre numeri separati da punti che indicano, rispettivamente, la versione *major*, la versione *minor e* la *patch*; per esempio: *1.4.12*. La versione *major* viene incrementata ogni volta che si apportano delle drastiche modifiche al software, rendendolo incompatibile con le versioni precedenti. La versione *minor* viene incrementata quando si modifica il codice in maniera minore, aggiungendo o modificando delle funzionalità in maniera compatibile con le versioni precedenti. Il numero di *patch* è incrementato ogni volta che si apportano delle modifiche o delle correzioni anche minime al sistema. La *major version zero* (*0.y.z*) è destinata allo sviluppo iniziale, quando il software non è ancora stabile e tutto può cambiare in ogni momento.

Prima ti ho detto che gli scienziati non sono infallibili. Non è una maldicenza: è la verità; gli scienziati sono i primi ad ammetterlo e questa è la loro forza, perché possono correggere i loro errori senza perdere di credibilità. Il Maestro Canaro voleva che questo fosse possibile anche per il C'hi+. Come scrisse alla fine della sua *Proposta per una metafisica open-source*:

> Se anche un giorno dovessi scoprire che gli elementi costituivi dell'Universo non si chiamano *spazioni*, ma *culturi* e fossi per ciò costretto a cambiare il nome della mia cosmogonia in *Culturismo*, io incrementerò di un'unità la *major-version* del mio progettto e andrò avanti. Non per ostinazione, né per idealismo, ma perché questa *metafisica-non-metafisica*, come l'ho definita prima, funziona: mi aiuta a decidere quale sia la cosa giusta da fare quando non è facile

> capire quale sia la cosa giusta da fare e mi aiuta ad affrontare i momenti difficili della vita, mia o altrui che sia, senza accettazioni per fede, ma basandomi solo su considerazioni di ordine logico. Inoltre, è una storia che non è ancora stata scritta e, allo stesso tempo, la storia che la nostra razza scrive da sempre. Mi sembra un motivo più che sufficiente.

Classi e oggetti

> Ceci n'est pas une |

La possibilità di definire nuovi tipi di dato grazie alle classi è la caratteristica principale del C++.

I linguaggi di programmazione "tradizionali", come il Cobol il Fortran o il Pascal, hanno un insieme limitato di tipi di dato: interi, numeri in virgola mobile, booleani, caratteri e stringhe.. giusto quello che serve a gestire una scheda anagrafica o un conto in banca. Il C e il Pascal hanno anche la possibilità di accorpare questi dati in strutture, enumerati o array, ma si tratta solo di contenitori, privi di logica interna. Inoltre, come hai visto, i dati all'interno di una `struct` sono accessibili a qualunque componente del programma, quindi, se li si modifica, va modificato anche il codice che li utilizza. Immagina di definire una struttura per la gestione dell'orario, che contenga tre interi, uno per le ore, uno per i minuti e uno per i secondi:

```
struct Orario {
    int h;
    int m;
    int s;
};
```

Per utilizzare questa struttura è necessario conoscerne il contenuto e il rapporto fra un valore e l'altro; in particolare, è necessario sapere (e ricordarsi):

- che la variabile `h` può contenere solo valori da 0 a 23;
- che il valore di `m` può contenere solo valori da 0 a 59;
- che il valore di `s` può contenere solo valori da 0 a 59;
- che se `s` supera il valore di 59, `m` va incrementato di 1;
- che se `m` supera il valore di 59, `h` va incrementato di 1;

Questo è l'opposto del *low coupling* di cui abbiamo parlato tempo fa, perché lega indissolubilmente una funzione alla struttura del dato che deve gestire. Per capirsi: una funzione di aggiornamento dei minuti dovrà essere qualcosa di simile a:

```
void aggiornaMinuti(struct Orario &o, int minuti)
{
    /** Incrementa il numero dei minuti */
    o.m += minuti;

    /** Se necessario, incrementa le ore */
    if(o.m >= 60) {

        o.m -= 60;
        o.h += 1;

        /** Se necessario, passa al giorno dopo */
        if(o.h >= 24) {
            o.h -= 24;
        }
    }
}
```

Se un giorno decidessimo di modificare la struttura `Orario`, dovremmo ricordarci di riscrivere anche questa funzione, adeguandola alle nuove caratteristiche della struttura, con dispendio di tempo e la possibilità di fare degli errori. Inoltre, nulla impedirebbe a un programmatore cialtrone di scrivere una funzione che non tiene minimamente conto del rapporto fra ore, minuti e secondi:

```
void incrementa_m(struct Orario &o, int minuti)
{
    o.m += minuti;
}
```

Se inseriamo queste due funzioni in un programma, otteniamo:

```
/**
 * @file classi-struttura-orario.cpp
 * Gestione dei dati membro di una struct.
 */

#include <iostream>
#include <iomanip>

using namespace std;

/** Dichiarazione di una struttura per gestire un orario */
struct Orario {
    int h;
    int m;
    int s;
};
```

```
/** Funzione per l'incremento dei minuti */
void aggiornaMinuti(struct Orario &o, int minuti)
{
    /** Incrementa il numero dei minuti */
    o.m += minuti;

    /** Se necessario, incrementa le ore */
    if(o.m >= 60) {

        o.m -= 60;
        o.h += 1;

        /** Se necessario, passa al giorno dopo */
        if(o.h >= 24) {
            o.h -= 24;
        }
    }
}

/*  Funzione cialtrona per l'incremento dei minuti */
void incrementa_m(struct Orario &o, int minuti)
{
    o.m += minuti;
}

int main()
{
    struct Orario ora;

    /** Definisce dei valori iniziali prossimi al cambio di data */
    ora.h = 23;
    ora.m = 45;
    ora.s = 00;

    /** Visualizza i valri iniziali */
    cout << setfill('0') << setw(2) << ora.h << ":"
         << setfill('0') << setw(2) << ora.m << ":"
         << setfill('0') << setw(2) << ora.s << endl;

    /** Richiama la funzione di aggiornamento */
    aggiornaMinuti(ora, 20);

    /** Visualizza i valori dopo l'aggiornamento */
    cout << setfill('0') << setw(2) << ora.h << ":"
         << setfill('0') << setw(2) << ora.m << ":"
         << setfill('0') << setw(2) << ora.s << endl;
```

```
    /** Reimposta i valori iniziali */
    ora.h = 23;
    ora.m = 45;
    ora.s = 00;

    /** Richiama la funzione cialtrona */
    incrementa_m(ora, 20);

    /** Visualizza i valori dopo l'aggiornamento */
    cout << setfill('0') << setw(2) << ora.h << ":"
         << setfill('0') << setw(2) << ora.m << ":"
         << setfill('0') << setw(2) << ora.s << endl;


    return 0;
}
```

Compilando ed eseguendo questo codice, ottieni:

```
> g++ src/cpp/classi-struttura-orario.cpp -o src/out/esempio
> ./src/out/esempio
23:45:00
00:05:00
23:65:00
```

Come puoi vedere, la prima funzione ha aggiornato i dati in maniera corretta, mentre la seconda ha prodotto un valore non valido senza alcuna possibilità di controllo da parte del programma.

Le variabili all'interno di una classe, sono dette *dati membro* o *attributi* della classe; le funzioni, invece, sono dette *funzioni membro* o *metodi*. Quando si crea una variabile di classe `X`, si dice che si: *istanzia* un *oggetto* di classe `X` o che si crea una *istanza* della classe. I dati e le funzioni membro di una classe sono direttamente accessibili alle funzioni membro della classe, ma per utilizzarli all'interno di funzioni esterne alla classe, si devono utilizzare gli operatori di selezione `.` e `->`. Il primo, detto *operatore di selezione diretta*, viene utilizzato con istanze della classe; il secondo, detto *operatore di selezione indiretta*, con puntatori ad esse:

```
/**
 * @file classi-punto.cpp
 * Accesso ai dati membro della classe.
 */

#include <iostream>
#include <iomanip>
```

```
using namespace std;

class Punto
{
public:

    /** Dichiara i dati membro della classe */
    int _x, _y;

    /**
     *  Le funzioni interne alla classe accedono ai
     *  dati membro con la sintassi ordinaria.
     */
    Punto(int x, int y) {
        _x = x;
        _y = y;
    }
} ;

int main(int argc, char** argv)
{
    /** Crea un oggetto di classe Punto */
    Punto p(5,6);

    /** Assegna l'istanza della classe al puntatore ptr */
    Punto *ptr = &p ;

    /**
     *  Le funzioni esterne alla classe accedono ai
     *  dati membro tramite gli operatori di selezione.
     */
    p._x    = 3 ;       // assegna un valore tramite l'oggetto
    ptr->_y = 2 ;       // assegna un valore tramite il puntatore

    //...
}
```

L'etichetta `public` che vedi all'inizio della dichiarazione della classe è un *specificatore di accesso* e serve a stabilire quali membri della classe siano accessibili a funzioni esterne e quali invece siano riservati in esclusiva alla classe stessa.

> il selvaggio non ama dire il suo nome o farsi fotografare, perché per mezzo del suo nome o del ritratto egli è accessibile, e può quindi ricevere danno da chi con questi mezzi è in grado di raggiungerlo.

Questa frase di Lucien Lévy-Bruhl si applica anche alle classi del C++. Sia gli

attributi che i metodi di una classe possono essere protetti da accessi o modifiche indebite grazie ai modificatori di accesso `private`, `protected` e `public`. I metodi o gli attributi dichiarati `private` sono accessibili solo alla classe stessa; quelli dichiarati come `protected` sono accessibili alla classe e a eventuali classi derivate; quelli dichiarati come `public` sono accessibili a qualunque elemento del programma. In mancanza di specifiche, tutti i dati e le funzioni di una classe verranno considerati:

- *privati*, nel caso di una classe;
- *pubblici*, nel caso di `struct` o `union`.

La visibilità dei dati membro di una `struct` può essere modificata con gli indicatori di accesso; i dati delle `union`, invece, possono essere solo pubblici.
Il C++ permette di suddividere la dichiarazione di una classe in quante sezioni si desidera e nella sequenza `private`, `public`, `protected` che si preferisce, ma un codice scritto in questo modo è sicuramente più difficile da leggere di uno in cui tutti i membri privati stanno da una parte e tutti quelli pubblici da un'altra. Quindi, a meno che tu non abbia delle buone ragioni per fare altrimenti (e ce ne potrebbero essere, nel caso di classi particolarmente complesse), cerca di raggruppare in tre sole sezioni `private`, `protected` e `public` tutte le funzioni e i dati membro con gli stessi attributi di accesso:

```
class Persona
{
 private:
    ...
 protected:
    ...
 public:
    ...
};
```

Questo tipo di ordinamento della dichiarazione, oltre a garantirti una maggiore leggibilità del codice, ti consentirà, se lo desideri, di omettere lo specificatore di accesso `private` iniziale (è la soluzione di default, ricordi?).
Mettiamo in pratica tutto ciò, convertendo in classe la struttura `Orario`:

```
class Orario {
private:
    int _h;
    int _m;
    int _s;
public:
    Orario() {
        _h = 0;
        _m = 0;
        _s = 0;
    }
```

```
};
```

La dichiarazione inizia con la parola-chiave `class`, seguìta dal nome della classe. Nel blocco di codice fra parentesi graffe che costituisce il corpo della classe, contiene i dati e le funzioni membro, accorpate per visibilità. In questo caso abbiamo messo prima i dati membro privati e poi quelli pubblici, ma avremmo potuto fare anche il contrario. Gli attributi `_h`, `_m` e `_s` compaiono dopo la parola-chiave `private` e saranno quindi visibili solo alle funzioni della classe stessa.
La funzione `Orario` compare dopo l'etichetta `public` e sarà accessibile per ciò a qualsiasi parte del programma. Questa funzione, che ha lo stesso nome della classe, è detta *costruttore* e viene richiamata ogni volta che si crea una variabile di tipo `Orario`. Il suo scopo è di inizializzare le variabili all'interno della classe, in questo caso, impostando tutti e tre i valori a 0. Ne parleremo fra poco.
L'ultima cosa che devi notare, nel codice qui sopra, è la presenza del carattere `;` alla fine del blocco di codice della classe, così come avviene per le `union` e le `struct`.

Quando dichiariamo una variabile di tipo primitivo come `int`, o `double`, il compilatore svolge automaticamente tutta una serie di operazioni atte ad allocare lo spazio di memoria necessario a contenerla e a inizializzarlo. Il compilatore, però, non sa come vada creata e inizializzata una variabile di tipo `Orario` ed è per questo che la classe dovrà definire delle *funzioni di gestione* che spieghino sia come creare una nuova variabile, che come distruggerla, se necessario. Le funzioni di gestione sono di due tipi: i *costruttori* e i *distruttori*.
I costruttori hanno alcune peculiarità che le distinguono dalle altre funzioni membro:

- hanno lo stesso nome della classe;
- non hanno un tipo di ritorno perché è implicito che ritornino una variabile della classe cui appartengono.

Una stessa classe può avere più costruttori; la classe `Orario`, per esempio, potrebbe avere un costruttore privo di parametri, che inizializzi ore, minuti e secondi a zero e un costruttore che permetta invece di assegnare valori specifici a ciascun attributo:

```
class Orario {
protected:
    int _h;
    int _m;
    int _s;
public:
    Orario() {
        _h = 0;
        _m = 0;
        _s = 0;
```

```
    }
    Orario(int h, int m, int s)
    : _h(h % 24), _m(m % 60), _s(s % 60) {
    }
};
```

La riga:

```
: _h(h % 24), _m(m % 60), _s(s % 60)
```

si chiama: *lista di inizializzazione* e ed equivale a scrivere:

```
_h = h % 24;
_m = m % 60;
_s = s % 60;
```

L'utilizzo dell'operatore modulo `%` è indispensabile, in questo caso, per evitare che siano assegnati valori non corretti alle variabili.
Quando definisci un costruttore, puoi usare indifferentemente l'una o l'altra sintassi o anche mischiarle, a seconda dei casi. Un modo più succinto di ottenere lo stesso risultato con un unico costruttore è di utilizzare dei valori di default per i parametri:

```
Orario(int h = 0, int m = 0, int s = 0)
: _h(h % 24), _m(m % 60), _s(s % 60) {
}
```

Alle volte, può essere utile definire un costruttore che crei delle nuove variabili della classe partendo da variabili esistenti, operando quindi una sorta di clonazione. Questo tipo di funzioni si chiamano: *costruttori di copia* o: *costruttori di inizializzazione* e richiedono come argomento un riferimento a una variabile della stessa classe:

```
/**
 * Dichiarazione del costruttore di copia
 * all'interno della classe.
 * Possiamo copiare il valore delle variabili
 * così com'è perché è già stato verificato
 * dal costruttore della variabile o1.
 */
Orario::Orario(const Orario& )
: _h(o._h), _m(o._m), _s(o._s) {
}

/** Utilizzo */
Orario o2 = o1;
```

Il costruttore di copia è un tipo di costruttore molto importante in quanto presiede alla maggior parte delle attività di inizializzazione di oggetti della classe

cui appartiene; per questa ragione, nel caso non venga definito dall'utente, è automaticamente generato dal compilatore.

Come è facile intuire, mentre il costruttore di una classe presiede alla creazione di nuove variabili, il distruttore si occupa della loro cancellazione. Non sempre è necessario definire un distruttore per una classe. Una variabile di tipo `Orario`, che contiene solo tre interi, probabilmente non avrà bisogno di un distruttore, mentre una variabile che faccia uso di memoria dinamica quasi sicuramente sì. Il perché risulta più chiaro se si esamina la cosa dal punto di vista del compilatore. Per creare una variabile di tipo `Orario` il compilatore deve allocare spazio per:

```
3 * sizeof(int);
```

Quando arriva il momento di distruggere la variabile, il compilatore non farà altro che liberare i `3 * sizeof(int)` byte successivi all'indirizzo dell'oggetto; un comportamento che in questo caso è corretto, ma che potrebbe dare rivelarsi disastroso con una classe come questa:

```
class Buffer
{
private:
    char* _dati;
    int   _size;
 public:
    Buffer(int size)
    : _size(size) {
        _dati = new char[_size];
    }
    ...
};
```

In mancanza di istruzioni specifiche, per distruggere una variabile di tipo `Buffer`, il compilatore libererà `sizeof(char*) + sizeof(int)` byte dopo il suo indirizzo di memoria, ma così facendo, distruggerà solo l'intero `_size` e il puntatore a char `_dati`, senza liberare l'area di memoria a cui quest'ultimo puntava. Questo, come sai, è un grave errore ed è necessario quindi aggiungere alla classe una funzione che lo istruisca in tal senso.
Come il costruttore, il distruttore di una classe non ha tipo di ritorno, ma mentre ci possono essere più costruttori per una stessa classe, il distruttore è sempre unico. Non ha mai parametri formali e il suo nome è uguale a quello della classe cui appartiene, preceduto da un carattere tilde `~`:

```
class Buffer
{
private:
    char* _dati;
    int   _size;
```

```
 public:
    Buffer(int size)
    : _size(size) {
        _dati = new char[_size];
    }
    ~Buffer() {
        delete [] _dati
    }
};
```

I distruttori possono essere chiamati in due modi:

- *implicitamente*, dal programma, ogni volta che un oggetto esce dal suo campo d'azione o, nel caso di oggetti con visibilità globale, al termine della funzione `main`;
- *esplicitamente*, dal codice, ma in questi casi dovrai specificare il loro nome per intero, anteponendo il nome della classe e l'operatore di risoluzione `::`, così come vedremo fra poco.

Attenzione, però: se a uscire dal campo d'azione è un puntatore, il distruttore della classe non viene richiamato automaticamente, perciò gli oggetti creati in maniera dinamica con l'operatore `new` dovranno sempre distrutti per mezzo dell'operatore `delete`.

Le funzioni membro devono essere dichiarate all'interno della dichiarazione della classe e possono essere definite sia dentro che fuori di essa. Definirle all'interno della dichiarazione della classe equivale a dichiararle inline Se invece le si definisce esternamente alla dichiarazione della classe, vanno identificate aggiungendo il nome della classe prima di quello della funzione, seguito dall'operatore di risoluzione:

```
/**
 * @file classi-classe-orario.cpp
 * Gestione dei dati membro di una classe.
 */

#include <iostream>
#include <iomanip>

using namespace std;

/** Dichiarazione della classe */
class Orario {
private:
    int _h;
    int _m;
```

```
    int _s;
public:
    Orario(int h = 0, int m = 0, int s = 0);
};

/** Definizione del costruttore della classe */
Orario::Orario(int h, int m, int s)
: _h(h % 24), _m(m % 60), _s(s % 60)
{
}

/** La funzione main non fa parte della classe */
int main()
{
    Orario ora;

    /** Questo codice darà errore */
    cout << setfill('0') << setw(2) << ora._h << ":"
         << setfill('0') << setw(2) << ora._m << ":"
         << setfill('0') << setw(2) << ora._s << endl;

    return 0;
}
```

Se compili questo codice, però, ottieni un errore: la funzione `main` può utilizzare il costruttore della classe `Orario` perché è dichiarato `public`, ma non può né leggere né modificare gli attributi definiti come `private`:

```
> g++ src/cpp/classi-classe-orario-1.cpp -o src/out/esempio
src/cpp/classi-classe-orario-1.cpp:34:44: error: '_h' is a protected member of 'Orario'
    cout << setfill('0') << setw(2) << ora._h << ":"
                                           ^
src/cpp/classi-classe-orario-1.cpp:14:9: note: declared protected here
    int _h;
        ^
src/cpp/classi-classe-orario-1.cpp:35:44: error: '_m' is a protected member of 'Orario'
         << setfill('0') << setw(2) << ora._m << ":"
                                           ^
src/cpp/classi-classe-orario-1.cpp:15:9: note: declared protected here
    int _m;
        ^
src/cpp/classi-classe-orario-1.cpp:36:44: error: '_s' is a protected member of 'Orario'
         << setfill('0') << setw(2) << ora._s << endl;
                                           ^
src/cpp/classi-classe-orario-1.cpp:16:9: note: declared protected here
    int _s;
        ^
```

```
3 errors generated.
```

Il C++ prevede due modi per rendere disponibili gli attributi di una classe anche alle funzioni esterne alla classe stessa:

- le classi o le funzioni `friend`.
- le *funzioni di interfaccia*;

Il modo apparentemente più rapido per accedere ai dati privati di una classe attraverso una funzione o una classe esterna è quello di dichiararle come `friend`. In virtù di ciò, la funzione o la classe acquisteranno una visibilità completa sui dati protetti:

```
class Orario {
private:

    /** Dati membro privati */
    int _h;
    int _m;
    int _s;

public:

    /** Costruttore della classe */
    Orario(int h = 0, int m = 0, int s = 0) ;

    /** Costruttore di copia inline */
    Orario(const Orario& o) {
        _h = o._h;
        _m = o._m;
        _s = o._s;
    }

    /** Funzioni di lettura  */
    int getH() { return _h; }
    int getM() { return _m; }
    int getS() { return _s; }

    /** Funzioni di scrittura */
    int setH(int h) { return _h = (h % 24); }
    int setM(int m) { return _m = (m % 60); }
    int setS(int s) { return _s = (s % 60); }

    /** Dichiarazione di una funzione friend */
    friend int aggiornaMinuti(Orario& o, int m);
    friend class Orologio();

};
```

Come ho detto, questa soluzione è solo apparentemente più rapida, perché tutta la logica di gestione dei dati della classe `Orario` dovrà essere replicata sia nella funzione `aggiornaMinuti` che nella classe `Orologio`. Inoltre, se in seguito dovessi apportare delle modifiche alla classe `Orario`, le stesse modifiche andranno riportate anche nelle funzioni delle classi `friend` che la utilizzano.
Un metodo più sicuro e più efficiente di gestire gli attributi privati di una classe consiste nel definire delle funzioni membro pubbliche che consentano un accesso controllato ai dati che si vogliono proteggere. Nel caso della classe `Orario`, ne occorrono sei: una per la lettura e una per la scrittura di ciascuno dei tre dati membro:

```
/** Funzioni di lettura */
int getH() { return _h; }
int getM() { return _m; }
int getS() { return _s; }

/** Funzioni di scrittura */
int setH(int h) { return _h = (h % 24); }
int setM(int m) { return _m = (m % 60); }
int setS(int s) { return _s = (s % 60); }
```

Ovviamente, puoi chiamare queste funzioni come preferisci, ma utilizzare i prefissi `get` e `set`, seguiti dal nome del parametro su cui operano rende più facile l'utilizzo della classe da parte di altri programmatori. È lo stesso motivo per cui aggiungo il carattere *underscore* davanti al nome dei dati membro delle classi, in modo che li si possa distinguere dai parametri delle funzioni che abbiano lo stesso nome:

```
return _h = (h % 24);
```

L'utilizzo della lista di inizializzazione, all'interno del costruttore, ti permette di utilizzare dei parametri che abbiano lo stesso nome dei dati membro della classe:

```
class Punto
{
    int x, y;
public:
    Punto(int x, int y)
    : x(x), y(y) {
    }
};
```

ma il fatto che qualcosa sia possibile non vuol dire che sia una buona scelta, come penso che tu abbia imparato, nel corso della tua vita.
Non sei nemmeno obbligato a dichiarare le funzioni di interfaccia come `inline`; l'ho fatto qui perché erano estremamente semplici, ma si dovrebbe evitare di aggiungere il codice delle funzioni all'interno della dichiarazione di una classe già di per sé complessa perché la rende più difficile da leggere. C'è anche chi

pensa che ciò sia sbagliato perché, se da un lato rende le cose più facili a chi scrive il codice, complica la vita di chi lo legge perché mischia ciò che la classe fa con il modo in cui lo fa. Io non sono del tutto d'accordo con questa affermazione perché alle volte è più comodo e rapido avere il codice delle funzioni all'interno della dichiarazione della classe, ma essendo un precetto che antepone il bene di tanti (i fruitori del codice) rispetto a quello del singolo (l'autore del codice), mi sono sentito in dovere di riferirtelo.
Così come abbiamo fatto per il costruttore della classe, potremmo unificare le funzioni di lettura e scrittura, utilizzando un parametro di default che determini il comportamento del programma:

```
int ore(int h = -1) {
    return _h = ((h != -1) ? _h = (h % 24) : h);
}
```

Questa sintassi è l'equivalente di:

```
int ore(int h = -1) {
    if(h != -1) {
        _h = (h % 24);
    }
    return _h;
}
```

Anche se meno evidente, è più comoda perché permette di tenere il codice su una sola riga e ti dà modo di fare un po' di pratica con gli operatori.
Questo tipo di funzioni, però, ha due difetti: limita i valori che puoi assegnare all'attributo e limita la granularità dei privilegi che puoi assegnare a chi utilizza la classe. Limita il numero di valori che puoi assegnare all'attributo, perché esclude il valore del parametro di default — cosa che non crea problemi in questo caso, dato che non esiste un'ora `-1`, ma che potrebbe farlo nel caso di una stringa con parametro di default nullo. Limita la granularità dei privilegi sulle funzioni, perché ti costringe a rendere pubbliche le funzioni di scrittura dei dati membro e questo, in certi casi potrebbe non essere saggio. Ti consiglio perciò di scrivere sempre due funzioni di interfaccia distinte per la lettura e la scrittura: sul momento ti sembrerà uno spreco di tempo, ma, a meno che il tuo programma non sia particolarmente banale, o prima o poi ti accorgerai di aver fatto la scelta corretta.

Ogni variabile di una determinata classe possiede delle copie dei dati membro, mentre le funzioni membro di una classe sono condivise da tutte le sue istanze. Per consentire al programma di sapere quale sia l'istanza che sta richiamando un determinato metodo, il compilatore aggiunge a ogni chiamata a funzione un parametro nascosto chiamato `this`, che punta all'istanza che ha richiesto la funzione. Il parametro `this`, anche se non dichiarato, può essere utilizzato nel corpo delle funzioni membro per riferirsi all'istanza corrente. Per esempio,

il costruttore di copia della classe `Orario` (così come qualsiasi altra funzione membro della classe) potrebbe essere riscritto così:

```
Orario(const Orario& o) {
    this->_h = o._h;
    this->_m = o._m;
    this->_s = o._s;
}
```

Le uniche funzioni membro che non possono fare uso del puntatore `this` sono quelle dichiarate come `static`.
Una classe può avere sia attributi che funzioni membro statiche. La particolarità di questi elementi è di non essere legati a una specifica variabile, ma di essere condivisi da tutte le istanze della classe; questo fa sì che abbiano un comportamento leggermente diverso da quello dei membri non statici:

- per inizializzarli all'interno della dichiarazione, li si deve dichiarare come `inline static`, altrimenti, devono essere inizializzati altrove nel programma, come un qualsiasi oggetto a visibilità globale;
- si può accedere ad essi, oltre che con i normali operatori di selezione, facendo riferimento alla classe stessa.

Cerco di chiarirti un po' le idee con un esempio:

```
/**
 * @file classi-static.cpp
 * Funzioni e dati membro statici di una classe.
 */

#include <iostream>

using namespace std;

/** Classe generica con membri statici */
class Contatore {
private:

    /** Definisce il dato statico */
    static int _nIstanze;

public:

    /** Il costruttore incrementa il numero di istanze */
    Contatore() {
        _nIstanze++;
    }

    /** Funzione di interfaccia statica */
```

```
    static int nIstanze() {
        return _nIstanze;
    }
};

/** Inizializza il membro statico  */
int Contatore::_nIstanze = 0;

int main()
{
    /** Crea la prima istanza della classe */
    Contatore c1;

    /** Richiama la funzione statica dall'oggetto */
    cout << "Da istanza c1: " << c1.nIstanze() << endl;

    /** Crea altre due istanze della classe */
    Contatore c2, c3;

    /**
     * Richiama la funzione statica dagli oggetti
     * e direttamente dalla classe
     */
    cout << "Da istanza c2: "<< c2.nIstanze()         << endl;
    cout << "Da istanza c3: "<< c3.nIstanze()         << endl;
    cout << "Dalla classe : "<< Contatore::nIstanze() << endl;

    return 0;
}
```

Se compili ed esegui questo codice, otterrai:

```
> g++ src/cpp/classi-static.cpp -o src/out/esempio
> src/out/esempio
Da istanza c1: 1
Da istanza c2: 3
Da istanza c3: 3
Dalla classe : 3
```

Come vedi, tutte le istanze della classe condividono lo stesso valore per il dato membro `nIstanze` e la funzione di interfaccia, dichiarata come `static`, può essere richiamata anche senza fare riferimento a un'istanza. Per questo motivo, se utilizzi il puntatore `this` all'interno di una classe statica:

```
static int nIstanze() {
    return this->_nIstanze;
}
```

ottieni l'errore di compilazione:

```
> g++ src/cpp/classi-static.cpp -o src/out/esempio
src/cpp/classi-static.cpp:26:16: error: invalid use of 'this'
 outside of a non-static member function
        return this->_nIstanze;
               ^
1 error generated.
```

perché, se la funzione fosse chiamata direttamente dalla classe, `this` non punterebbe ad alcun oggetto.

L'ultima cosa di cui dobbiamo parlare, sono le *classi anonime*, un tipo particolare di classe che, come dice il nome (perdonami il gioco di parole), non hanno nome e per ciò non possono avere né un costruttore né un distruttore e non possono essere utilizzate né come parametri né come valori di ritorno delle funzioni. L'unico modo per dichiarare un oggetto con classe anonima è di aggiungerlo alla dichiarazione della classe stessa:

```
class
{
...
} obj;
```

Questo codice dichiara allo stesso tempo la classe e la sua unica istanza, la variabile globale `obj`.
Quando il Maestro Canaro provò a fare il *porting* dell'Universo in C++, avrebbe voluto utilizzare una classe anonima per la variabile `Dio`, perché, priva di un costruttore e istanziata dalla sua stessa classe, quella variabile sarebbe stata visibile in tutto il codice, ma nessuna parte del programma ne avrebbe potuta generare un'altra:

```
/**
 * @file classi-dio.cpp
 * Dichiarazione della classe Dio.
 */

#include <iostream>
#include <list>


using namespace std;

// Classe astratta da utilizzare come base
// per tutti gli elementi del Creato.
class Creatura {
public:
```

```
   // Funzione virtuale pura di verifica.
   virtual bool isGood() = 0;

};

// Classi derivate per la gestione degli
// elementi del Creato.
class Mare   : public Creatura {
    bool isGood() { return true; };
};
class Uomo   : public Creatura {
    bool isGood();
};
class Donna  : public Creatura {};
class Padre  : public Uomo     {};
class Figlio : public Uomo     {};
class Popolo : public list<Creatura> {};

class  {
private:

    // Funzioni membro per la generazione degli elementi
    // del Creato, accessibili solo alla classe.
    Creatura* creaLuce();
    Creatura* creaStelle();
    Creatura* creaAcquaTerra();
    Creatura* creaPiante();
    Creatura* creaSoleLuna();
    Creatura* creaAnimali();

    // Funzioni per la generazione degli umani.
    Uomo&  creaUomo();
    Donna& creaDonna(Uomo& adamo);

    // Interruzione per il settimo giorno.
    void shabat();

    // Funzioni di interfaccia con gli umani.
    bool popEden(Uomo& adamp, Donna& eva);
    bool printComandamenti(ostream& tavole);
    bool checkFede(Padre& abramo, Figlio& isacco);
    bool splitAcque(Mare& marRosso, Popolo& ebrei);

protected:

    // Funzione membro accessibile anche alle
```

```
    // classi figlio.
    bool donaVita(Uomo& lazzaro);

public:

     // Funzione membro pubblica.
     // Torna il numero di preghiere da recitare.
    int rimettiPeccati(Creatura* fedele);

} Dio;
```

L'idea, in sé, era buona, ma venne abbandonata quando si trattò di definire gli attributi e i metodi della classe. Dio, infatti, ha *tutti* gli attributi immaginabili e ciascuno di essi ha valore infinito. Allo stesso modo, essendo onnipotente, deve avere delle funzioni membro per portare a termine *tutte* le possibili azioni e il codice di queste funzioni, utilizzando una classe anonima, avrebbe dovuto essere definito all'interno della dichiarazione della classe, perché, mancando un nome, non lo si sarebbe potuto definire esternamente:

```
bool ...::checkFede(Padre& abramo, Figlio& isacco)
{

}
```

È un peccato che Platone non sia vissuto duemilaquattrocentoundici anni, perché avrebbe certamente apprezzato l'affinità delle classi del C++ con le sue teorizzazioni riguardo le *idee* e le *forme*. In questo codice, possiamo considerare la dichiarazione della classe come l'*idea* del pesce, mentre l'istanza ne è la *forma*:

```
/**
 * @file classi-pesce.cpp
 * Definizione della classe Pesce.
 */

#include <iostream>
#include <iomanip>

using namespace std;

/**
 *  Dichiarazione della classe: idea
 */
class Pesce {
private:

    /** Dati membro privati */
```

```
    string _specie;
    bool   _commestibile;

public:

    /** Costruttore della classe inline */
    Pesce(const char* specie, bool commestibile)
    : _specie(specie), _commestibile(commestibile)
    {
    }

    /** Costruttore di copia inline */
    Pesce(const Pesce& p) {
        this->_specie       = p._specie;
        this->_commestibile = p._commestibile;
    }

    /** Distruttore inline */
    ~Pesce() {
    }

    /** Funzioni di interfaccia */
    const char* getSpecie() { return _specie.c_str(); }
    bool isCommestibile()   { return _commestibile;   }

};


/**
 *  Istanza della classe: forma
 */
int main()
{
    Pesce pesce("spigola", true);

    cout << pesce.getSpecie() << ": "
         << (pesce.isCommestibile() ? "sì" : "no")
         << endl;

    return 0;
}
```

Come abbiamo detto parlando del preprocessore, però, la parola *pesce* può avere diversi valori, a seconda di chi la utilizza, quindi, la dichiarazione/idea della classe `Pesce` varierà a seconda dell'utilizzo che se ne deve fare. Per esempio, sapere se un pesce sia commestibile o no è determinante per un pescatore o per

un ecologista, ma potrebbe non esserlo per un biologo marino. Al contrario, il tipo di scheletro o il sistema di respirazione, rilevanti per un biologo, sono del tutto irrilevanti per un pescivendolo, a cui invece interesseranno sicuramente il prezzo al chilo, la data di cattura e il tipo di conservazione applicato. Di questo, parleremo nelle prossime lezioni; adesso dobbiamo tornare su una questione che avevamo lasciato in sospeso, ovvero il precetto:

> Amiamo ciò che ci ucciderà (se tutto va bene)

Abbiamo visto che l'Amore è una forza allo stesso tempo gravitazionale ed entropica, perché unisce gli individui, ma allo stesso tempo li porta a riprodursi in forme differenti.
In un certo senso, possiamo considerare l'Amore come il "costruttore" delle nostre istanze, perché genera le condizioni che spingono i nostri genitori a incontrarsi e ad accoppiarsi e soprattutto li spinge a restare insieme dopo l'accoppiamento.
La monogamia non è una costante, anzi, in natura esistono quattro modi differenti di gestire la prole e John Maynard Smith li ha catalogati in base alla specie animale che le adotta:

Anitra	il maschio abbandona, la femmina alleva
Spinarello	la femmina abbandona, il maschio alleva
Moscerino	entrambi i genitori abbandonano
Gibbone	entrambi i genitori allevano

Noi, per lo più, ci comportiamo come i gibboni, anche se è un comportamento che conviene principalmente alle femmine. Da un punto di vista strettamente statistico, un maschio avrebbe più probabilità di tramandare il suo DNA se fecondasse più compagne. È lo stesso motivo per cui, nel nostro campo, si creano delle copie dei proprii dati in *server-farm* diverse (e distanti) da quella originale, in modo che se il sito principale va a fuoco o se viene colpito da un meteorite, i dati non vadano persi. L'Amore, invece, spinge il *webmaster* a tenere i suoi dati in una sola *server-farm*, accudendoli e proteggendoli personalmente, per preservarne il contenuto informativo.
Ciascuno di noi è l'istanza di una classe che è stata chiamata a vivere per svolgere un determinato compito. Veniamo generati, assolviamo il nostro compito e poi, così come gli oggetti di un programma, dobbiamo essere rimossi per non occupare inutilmente delle risorse del sistema. Per essere certi che questo avverrà, c'è bisogno di un distruttore che termini la nostra esistenza nel momento opportuno. L'Amore può servire anche a questo: così come ha generato nei nostri genitori l'interesse necessario a causare la nostra nascita, genera in noi un interesse che causa le condizioni necessarie alla nostra morte. Questo interesse può applicarsi a qualunque cosa — una sostanza, un'attività, un luogo, una o più persone — e deve essere superiore all'interesse che l'individuo ha nei confronti della sua stessa esistenza.
Capisci bene che questo è un aspetto potenzialmente rischioso della nostra dottrina, perché potrebbe giustificare delle forme di auto-indulgenza nei confronti

di sostanze o attività dannose come l'abuso di droga o alcol. La morte di Robert Capa, sopravvissuto allo sbarco in Normandia e ucciso, dieci anni dopo, da una mina anti-uomo mentre fotografava dei soldati sul delta del Fiume Rosso, è un esempio inequivocabile di questo precetto, ma come possiamo sapere se la morte di Jimi Hendrix, Janis Joplin o Jim Morrison, sia stata ciò che doveva essere, o non sia stata, al contrario, la conseguenza di una scelta drammaticamente errata? Soprattutto, come possiamo evitare che questo precetto non diventi il pretesto per altre scelte errate?
Ho scelto le tre "J" — Jim, Jimi e Janis — perché il Maestro Canaro pensava che, nel loro caso, la morte fosse ciò a cui erano destinati:

> Capisci, avevano tutti ventisette anni e poi quella "J" che ricorre nel nome.. non credo fosse un caso. "*Muß es sein? Es muß sein!*", come direbbe Beethoven. So cosa stai pensando: che sto cercando di definire un dogma per giustificare una mia speranza, ma non è così. La teoria del "27 Club", anche se include Amy Winehouse e Robert Johnson, lascia dolorosamente fuori il Elvis e Andrea Pazienza, che io ho amato molto di più.

D'altro canto, non sempre le cose vanno come previsto e può accadere che per una scelta errata o per paura, una "istanza" devii dal suo cammino anticipando o ritardando la propria fine. Questo perché il nostro agire è preordinato, ma non è obbligato: siamo programmati per fare bene qualcosa e meno bene quacos'altro, ma siamo liberi di scegliere cosa fare.
Molti si sono chiesti se le nostre azioni siano predestinate o se esista il libero arbitrio; la risposta è: sì, ma in tempi diversi. Riprendendo il paragone con le classi, la predestinazione è nella dichiarazione, mentre il libero arbitrio è proprio delle istanze. L'insieme dei tuoi attributi e dei tuoi metodi è predefinito: puoi saltare, ma non puoi volare; puoi nuotare, ma non puoi restare a lungo in immersione; il tuo sangue ha un flusso ben definito e così pure il tuo cibo. D'altro canto, quello che deciderai di fare dipende da te: puoi decidere di attenerti ai limiti che ti sono stati imposti o cercare di sorpassarli, creandoti delle ali artificiali per volare o dei respiratori per andare sott'acqua. Attento, però: questo non farà di te né un uccello né un pesce. Puoi superare i limiti imposti dalla tua classe, ma non la puoi cambiare.
Non è difficile conciliare il libero arbitrio con l'onniscenza e l'onnipotenza di Dio; la risposta è ovvia, se non fai distinzione fra artefatto e artefice. Con buona pace di Einstein, Dio gioca davvero a dadi con l'Universo; lo sbaglio è pensare che il risultato di un tiro di dadi derivi da fattori casuali. Quando si lanciano dei dadi, il risultato finale varia in base a due ordini di fattori, uno facilmente prevedibile e uno difficilmente prevedibile. Le possibili combinazioni dei dadi sono facilmente prevedibili, noto il numero delle facce e i valori che vi sono impressi. La combinazione che verrà effettivamente prodotta da un certo lancio di dadi è altrettanto deterministica, ma dipende da fattori molto più complessi, come il tempo per cui li si è agitati, della loro posizione al momento del lancio

o l'angolo di impatto con il piano. Così come è impossibile che un lancio di dadi nel Backgammon produca un valore superiore a dodici, è impossibile che un dado, lanciato con una certa energia in una certa direzione adotti una traiettoria diversa da quella che gli impongono le Leggi della fisica. Anche se non riusciamo a prevederla, non vuol dire che sia casuale.
Allo stesso modo, quando l'Uno primigenio "lancia" la sua energia nell'Universo, è *onniscente*, perché conosce tutte le scelte che in precedenza si sono rivelate corrette e quelle che invece hanno prodotto del dolore; è *benevolo*, perché spera di ottenere il miglior risultato possibile, evitando di ripetere gli sbagli già fatti; è *onnipotente*, perché può potenzialmente produrre tutte le possibili permutazioni dell'esistenza dovute all'interazione energia/spazioni. Malgrado ciò, non sa quale di quelle permutazioni avrà luogo e non lo vuole nemmeno sapere. Auspica che il nuovo ciclo di esistenza sia migliore dei precedenti, ma non desidera che avvenga un certo evento o che non avvenga un altro, perché il desiderio lo renderebbe vulnerabile alle lusinghe dell'Annosa Dicotomia.
Un famoso velista, una volta disse:

> Quando sei in regata e non c'è vento, non lo andare a cercare.
> Metti la prua nella direzione giusta e aspetta: il vento arriverà.

L'Uno si comporta in maniera simile: ligio al precetto del *Wu Wei* taoista, pone le condizioni necessarie per il ripetersi degli eventi che si sono rivelati benefici, ma non li impone. Scrive l'analisi del sistema, ma lascia che siano i programmatori a scrivere il codice, anche se sa che faranno certamente degli errori. Definisce delle regole, ma lascia le sue istanze libere di trasgredirle, perché sa che l'evoluzione è sempre frutto di un errore venuto male, di qualcosa che non sarebbe dovuto essere così e invece così è meglio.
Le regole non devono essere una rete che imprigiona e immobilizza, ma una rete che salva e sostiene, così come il "religare" delle religioni non deve essere un legame che impastoia, ma che sorregge. Le regole che definiscono e quindi limitano la nostra esistenza sono come un edificio che abbia una struttura in cemento armato e dei muri in cartongesso. I muri interni possono essere abbattuti o modificati, se necessario, ma i pilastri e le travi devono essere lasciati al loro posto. Similmente, la modifica delle regole può essere benefica, ma deve essere permessa solo a chi le conosce bene perché un carpentiere maldestro potrebbe - per errore o per stupidità - rimuovere uno dei pilastri portanti mettendo in pericolo la solidità dell'edificio.
Per questi motivi, la modifica delle regole non può essere un'attività ammessa da chi le ha promulgate, anche se ne riconosce l'utilità, ma dev'essere un'attività apparentemente clandestina, svolta da elementi sacrificabili, che possano fungere da capri espiatorii se qualcosa va male.
Non fu per ingenuità, che l'Altissimo concesse a Iblīs una proroga alla sua punizione e non fu un caso se il Maestro Canaro venne aggiunto al gruppo degli angeli caduti, dopo che, vittima dell'Annosa Dicotomia, cercò di contravvenire alle regole definite dall'Analista. Se rifiuti le dissonanze, tutt'al più, puoi suonare il Blues; con le dissonanze, hai il Jazz.

Così come l'immagine di una stampa litografica o di una serigrafia esiste sia nella matrice che nella copia, ciascuno di noi ha due livelli di esistenza. Uno è ideale, simile alla dichiarazione della classe e definisce quale sia il nostro ruolo nell'esistenza: ciò che *possiamo* fare, ciò che *non possiamo* fare e ciò che *dovremmo* fare. L'altro livello è la nostra manifestazione reale, dovuta all'interazione dell'energia dell'Uno con gli spazioni. Questo livello è assimilabile all'istanza di una classe, che mette in atto ciò che nella dichiarazione era solo potenziale: *the hearts that break, the mess we make*, come dice la canzone.
La nostra entità ideale è unica e costante, mentre la nostra manifestazione fisica è mutevole: come sai, uno stesso oggetto può essere allocato in aree differenti di memoria, in successive esecuzioni di un programma, così come una stessa stampa può essere riprodotta su supporti diversi. In questo ciclo di esistenza, il Maestro Canaro e il cane Lele sono stati un umano e un cane che correvano sulle colline intorno al lago di Bracciano, ma in altre esistenze potrebbero essere — o essere stati — altre persone e altri animali, in altri luoghi o addirittura in altri pianeti. Il nostro livello ideale, infatti, non stabilisce cosa dobbiamo essere, ma quale debba essere il nostro contributo all'economia dell'Universo; la forma che assumiamo o il luogo in cui ci manifestiamo sono del tutto incidentali. Per il C'hi++, come per l'Induismo, la frase: "Cogito ergo sum" di Cartesio è insensata, perché ciò che cogita è l'istanza, che è transeunte. Ciò che siamo realmente, la nostra essenza, si manifesta in ciò che facciamo istintivamente.

L'ereditarietà

> Una classe ma può ripudiare una classe figlia, se è cattiva, ma una casse figlia non può ripudiare la sua classe padre perché gli deve più di quanto non sia in grado di pagargli e in particolare gli deve l'esistenza.

L'ereditariaretà, ovvero la possibilità di creare delle genealogie di classi, è la caratteristica principale del C++.

Come abbiamo visto nella lezione precedente, una ipotetica classe `Pesce` dovrà avere attributi differenti a seconda dell'utilizzo che se ne deve fare. In un linguaggio come il *C*, che non permette l'ereditariaretà, quindi, si dovranno prevedere due strutture di dati differenti per ciascun caso:

```
enum Acqua  { dolce, salata };
enum Sesso  { maschio, femmina };
enum Colore { rosso, blu, verde, argento };

struct PesceAlimentare {
```

```
    Sesso  _sesso;
    char*  _specie;
    float  _prezzo;
    time_t _data_cattura;
    int    _peso;
    bool   _commestibile;
    char*  _area_pesca;
    ...
};

struct  PesceAcquario {
    Sesso  _sesso;
    char*  _specie;
    float  _prezzo;
    time_t _data_acquisto;
    Colore _colore;
    Acqua  _acqua;
    ...
};
```

e funzioni differenti per la gestione dei dati:

```
void gestisciPesceAlimentare(struct PesceAlimentare& pesce);
void gestisciPesceAcquario(struct PesceEcologista& pesce);
```

Questo vuol dire che se hai già scritto (e verificato, corretto e collaudato) un programma di gestione per un pescivendolo, per poterlo trasformare in un programma di gestione per la vendita di pesci da acquario, dovrai riscrivere (e ri-verificare, ri-correggere e collaudare) tutto il codice, anche se parte dei dati da considerare sono gli stessi.
L'ereditariaretà, al contrario, ti permette di isolare in una classe le caratteristiche comuni a tutti e due i contesti e di *derivare* da questa classe di base due classi specializzate:

```
/**
 * @file ereditarieta-singola.cpp
 * Esempio di ereditarietà singola.
 */

#include <iostream>
#include <string>
#include <ctime>

using namespace std;

enum Acqua  { dolce, salata };
enum Sesso  { maschio, femmina };
enum Colore { rosso, blu, verde, argento };
```

```
class Pesce {
private:

    static int _n_pesci;

protected:

    /** Dati membro privati */
    Sesso  _sesso;
    float  _prezzo;
    string _specie;
    bool   _commestibile;

public:

    /** Costruttore della classe inline */
    Pesce(Sesso sesso, float prezzo, const char* specie)
    : _sesso(sesso), _prezzo(prezzo), _specie(specie)
    {
        _n_pesci++;
    }

    /** Distruttore inline */
    ~Pesce() {
        _n_pesci--;
    }

    /** Funzioni di interfaccia */
    void setSesso(Sesso sesso);
    void setPrezzo(float prezzo);
    void setSpecie(const char* specie);

    /** Funzioni di interfaccia inline */
    string getSpecie() { return _specie.c_str(); }
    float  getPrezzo() { return _prezzo;  }
    Sesso  getSesso()  { return _sesso; }
    bool   getCommestibile() { return _commestibile; }
    void   setCommestibile(bool commestibile) {
        _commestibile = commestibile;
    }

};

int Pesce::_n_pesci = 0 ;
```

```
class PesceAlimentare : public Pesce {
private:

    /** Dati privati della classe */
    time_t _data_cattura;
    string _area_pesca;

public:

    /** Costruttore della classe inline */
    PesceAlimentare(Sesso sesso, float prezzo, const char* specie)
    : Pesce(sesso, prezzo, specie)
    {
        /** Auspicabilmente.. */
        _commestibile = true;
    }

    //...
};

class PesceAcqario : public Pesce {
private:

    /** Dati privati della classe */
    time_t _data_acquisto;
    Colore _colore;
    Acqua  _acqua;

public:

    /** Costruttore della classe inline */
    PesceAcqario(Sesso sesso, float prezzo, const char* specie)
    : Pesce(sesso, prezzo, specie)
    {
    }

    //...
};
```

Questo approccio ha due lati positivi: il primo è che non sarà necessario ripetere le fasi di test, debug e collaudo per le funzioni comuni ai due sistemi, perché saranno state già verificate durante lo sviluppo del primo sistema; il secondo è che, riutilizzando parte del codice, sarà possibile identificare e correggere eventuali errori sfuggiti alla prima fase di test o migliorare il comportamento delle funzioni comuni, con benefici per entrambi i sistemi.

L'ereditarietà, nel C++, può essere o *singola* o *multipla*, a seconda che la nuova classe erediti le caratteristiche da una o più classi preesistenti:

```
class Figlio : public Mamma
{
    /** Ereditarietà singola */
};

class Figlio : public Mamma, private Papa
{
    /** Ereditarietà singola */
};
```

Al contrario, non è permesso (né sensato) che una classe erediti due volte dalla stessa classe base:

```
class Errore : public Mamma, public Mamma
{
    ...
};
```

Le classi `Mamma` e `Papa` possono essere definite: *classi base* o *sottoclassi* o *classi fondamentali* o *sotto-tipi*; la classe `Figlio` può essere chiamata o *classe derivata* o *superclasse* o *supertipo*.
L'istruzione:

```
class PesceAlimentare : public Pesce {
    ...
};
```

dichiara la classe `PesceAlimentare` come classe derivata dalla classe `Pesce`. Lo specificatore di accesso fra i nomi delle due classi definisce la visibilità dei dati della classe base all'interno della classe figlia:

```
class B : public A
{
    // tutti i membri di A mantengono
    // in B la loro visibilità originale
};

class B : protected A
{
    // tutti i membri public di A diventano membri
    // protected di B; i membri protected e private
    // mantengono la loro visibilità originale
};

class B : private A
{
```

```
    // tutti i membri di A, quale che sia la loro
    // visibilità, diventano membri private di B;
};
```

In mancanza di un qualificatore di accesso, il compilatore considera privati tutti i dati di una classe dichiarata con la parola chiave `class` e pubblici tutti i dati di una classe dichiarata con la parola chiave `struct`:

```
/**
 * @file ereditarieta-accesso.cpp
 * Accesso ai membri della classe base dalle classi derivate.
 */

/** Classe base, dati membro pubblici */
struct Mamma {
    int a;
};

/** Classe derivata come struct: dati per default pubblici */
struct Figlia : Mamma {
    int b;
};

/** Classe derivata come class: dati per default privati */
class Figlio : Mamma {
    int c;
};

/** Funzione generica, non friend delle classi */
void funz(Figlia& figlia, Figlio& figlio)
{
    figlia.a++;
    figlio.a++;
}
```

Se compili questo codice, ottieni un messaggio di errore:

```
> g++ src/cpp/ereditarieta-accesso.cpp -c -o src/out/esempio
src/cpp/ereditarieta-accesso.cpp:25:5: error: cannot cast 'Figlio'
    to its private base class
      'Mamma'
    figlio.a++;
    ^
src/cpp/ereditarieta-accesso.cpp:17:16: note: implicitly declared
    private here
class Figlio : Mamma {
               ^~~~~
src/cpp/ereditarieta-accesso.cpp:25:12: error: 'a' is a private
```

```
    member of 'Mamma'
    figlio.a++;
           ^
src/cpp/ereditarieta-accesso.cpp:17:16: note: constrained by
    implicitly private inheritance here
class Figlio : Mamma {
               ^~~~~
src/cpp/ereditarieta-accesso.cpp:8:9: note: member is declared
    here
    int a;
        ^
2 errors generated.
```

Questa è la dichiarazione del costruttore della classe derivata `PesceAlimentare`:

```
PesceAlimentare(Sesso sesso, float prezzo, const char* specie)
: Pesce(sesso, prezzo, specie) {
    _commestibile = true;
}
```

La seconda linea è la *lista di inizializzazione* della classe e contiene i costruttori delle classi base. Quando si instanzia un oggetto di classe derivata, il sistema richiama per prima cosa i costruttori delle classi base e poi quello della classe figlia. In questo modo, il costruttore della classe derivata ha la certezza di lavorare su dei dati membro correttamente inizializzati.
L'utilizzo del costruttore delle classi base per l'inizializzazione dei dati comuni è necessario per due motivi: il primo è che parte dei dati delle classe base potrebbero essere `private` e quindi inaccessibili alla classe derivata; il secondo motivo è che in questo modo si ottiene un low coupling fra classe base e classe derivata e, se si dovesse modificare l'implementazione interna del costruttore della classe base (mantenendo invariata l'interfaccia), non ci sarebbe bisogno di dover modificare il codice delle sue classi derivate.
L'ordine in cui i costruttori delle classi base sono chiamati durante l'inizializzazione dell'oggetto dipende dall'ordine in cui compaiono nel costruttore della classe figlia. Lo vediamo con un altro esempio, un po' più complesso del precedente, che mostra anche il funzionamento dei dati e delle funzioni membro statiche:

```
/**
 * @file ereditarieta-multipla.cpp
 * Esempio di ereditarierà multipla.
 */

#include <iostream>
#include <string>
#include <ctime>
```

```
using namespace std;

enum Acqua  { dolce, salata };
enum Sesso  { maschio, femmina, boh};
enum Colore { rosso, blu, verde, argento };

/**
 * Dichiarazione della classe Articolo
 */
class Articolo {
private:

    static int _n_articoli;

protected:

    /** Dati membro privati */
    float  _prezzo;

public:

    /** Costruttore della classe inline */
    Articolo(float prezzo = 0.0)
    : _prezzo(prezzo)
    {
        _n_articoli++;
        cout << "costruttore Articolo" << endl;
    }

    /** Distruttore inline */
    ~Articolo() {
        _n_articoli--;
        cout << "distruttore Articolo" << endl;
    }

    /** Funzioni di interfaccia */
    void setPrezzo(float prezzo);

    /** Funzione di interfaccia statica */
    static int getIstanze() { return _n_articoli; }

    /** Funzioni di interfaccia inline */
    float getPrezzo()     { return _prezzo;     }
    const char* getTipo() { return "articolo";  }
};
```

```
/** Inizializza il dato membro statico */
int Articolo::_n_articoli = 0 ;

/**
 * Dichiarazione della classe Pesce
 */
class Pesce {
private:

    static int _n_pesci;

protected:

    /** Dati membro privati */
    Sesso  _sesso;
    string _specie;
    bool   _commestibile;

public:

    /** Costruttore della classe inline */
    Pesce(Sesso sesso = boh, const char* specie = "ignota")
    : _sesso(sesso), _specie(specie)
    {
        _n_pesci++;
        cout << "costruttore Pesce" << endl;
    }

    /** Distruttore inline */
    ~Pesce() {
        _n_pesci--;
        cout << "distruttore Pesce" << endl;
    }

    /** Funzioni di interfaccia */
    void setSesso(Sesso sesso);
    void setSpecie(const char* specie);

    /** Funzione di interfaccia statica */
    static int getIstanze()      { return _n_pesci; }

    /** Funzioni di interfaccia inline */
    string getSpecie()       { return _specie.c_str(); }
    Sesso  getSesso()        { return _sesso; }
    bool   getCommestibile() { return _commestibile; }
```

```
    void   setCommestibile(bool commestibile) {
        _commestibile = commestibile;
    }
};

/** Inizializza il dato membro statico */
int Pesce::_n_pesci = 0 ;

/**
 * Dichiarazione della classe PesceAlimentare
 */
class PesceAlimentare
: public Pesce, public Articolo {
private:

    /** Dati privati della classe */
    time_t _data_cattura;
    string _area_pesca;

public:

    /** Costruttore della classe inline */
    PesceAlimentare(Sesso sesso, float prezzo, const char* specie)
    : Articolo(prezzo)
    , Pesce(sesso, specie)
    {
        /** Auspicabilmente.. */
        _commestibile = true;
        cout << "costruttore PesceAlimentare" << endl;
    }

    /** Distruttore inline */
    ~PesceAlimentare() {
        cout << "distruttore PesceAlimentare" << endl;
    }

    /** Ridefinizione della funzione della classe base */
    const char* getTipo() { return "pesce alimentare"; }

    //...
};

class PesceAcquario
: public Articolo, public Pesce {
private:
```

```
    /** Dati privati della classe */
    time_t _data_acquisto;
    Colore _colore;
    Acqua  _acqua;

public:

    /** Costruttore della classe inline */
    PesceAcquario(Sesso sesso, float prezzo, const char* specie)
    : Pesce(sesso, specie)
    , Articolo(prezzo)
    {
        cout << "costruttore PesceAcquario" << endl;
    }

    /** Distruttore inline */
    ~PesceAcquario() {
        cout << "distruttore PesceAcquario" << endl;
    }

    /** Ridefinizione della funzione della classe base */
    const char* getTipo() { return "pesce acquario"; }

    //...
};

int main(int argc, char** argv)
{
    /** Crea le due istanze delle classi */
    PesceAcquario pesce1(maschio, 2.5, "Paracheiredon");
    PesceAlimentare pesce2(femmina, 25.4, "Dicentrarchus labrax");

    /** Richiama le funzioni di interfaccia */
    cout <<  pesce1.getTipo() << ": " << pesce1.getSpecie()
         << endl;
    cout <<  pesce2.getTipo() << ": " << pesce2.getSpecie()
         << endl;

    /** Richiama la funzione della classe base */
    cout << "classe base: " << pesce1.Articolo::getTipo() << endl;

    /** Richiama le funzioni di interfaccia statiche */
    cout <<  "articoli creati: " << Articolo::getIstanze()
         << endl;
    cout <<  "pesci creati:    " << Pesce::getIstanze()
         << endl;
```

```
    return 0;
}
```

La differenza con il codice precedente è che in questo caso abbiamo isolato i dati relativi al costo del pesce in una classe separata e che le classi derivate ereditano non più da una classe base, ma da due.
Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/ereditarieta-multipla.cpp -o src/out/esempio
> src/out/esempio
costruttore Pesce
costruttore Articolo
costruttore PesceAlimentare
costruttore Articolo
costruttore Pesce
costruttore PesceAcquario
pesce acquario: Paracheiredon
pesce alimentare: Dicentrarchus labrax
classe base: articolo
articoli creati: 2
pesci creati:    2
distruttore PesceAcquario
distruttore Pesce
distruttore Articolo
distruttore PesceAlimentare
distruttore Articolo
distruttore Pesce
```

Come vedi, l'ordine di chiamata dei costruttori delle classi base rispecchia quello in cui sono elencate nella lista di inizializzazione, mentre quello dei distruttori è invertito. Se il costruttore della classe derivata non specifica un ordine di chiamata per i costruttori delle classi base, l'ordine di chiamata è dato dall'ordine in cui le classi base compaiono nella dichiarazione della classe figlio:

```
/**
 * @file ereditarieta-ordine-costruttori.cpp
 * Ordine di chiamata dei costruttori nell'ereditarierà multipla.
 */

#include <iostream>
#include <string>
#include <ctime>

using namespace std;

class Mamma {
public:
```

```
    Mamma() {
        cout << "costruttore Mamma" << endl;
    }
};

class Padre {
public:
    Padre() {
        cout << "costruttore Padre" << endl;
    }
};

class Figlio
: public Padre, public Mamma {
public:
    Figlio() {
        cout << "costruttore Figlio" << endl;
    }
};

class Figlia
: public Mamma, public Padre {
public:
    Figlia() {
        cout << "costruttore Figlia" << endl;
    }
};


int main(int argc, char** argv)
{
    Figlio Hansel;
    Figlia Gretel;

    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/ereditarieta-ordine-costruttori.cpp -o src/out/esempio
> src/out/esempio
costruttore Padre
costruttore Mamma
costruttore Figlio
costruttore Mamma
costruttore Padre
costruttore Figlia
```

Un'altra cosa da notare, in questo codice, è che la funzione `getTipo` è presente sia nella classe base `Articolo` che nelle due classi derivate `PesceAlimentare` e `PesceAcquario`. Per questo motivo, quando si richiama `getTipo` da un'istanza delle due classi derivate, come in:

```
cout <<  pesce1.getTipo() << ": " << pesce1.getSpecie()
cout <<  pesce2.getTipo() << ": " << pesce2.getSpecie()
```

il valore tornato è quello della funzione della classe figlia. Per ottenere il valore della classe base, dobbiamo specificarne il nome nell'istruzione, come in:

```
cout << "classe base: " << pesce1.Articolo::getTipo() << endl;
```

Complichiamo un po' le cose. Immagina che una classe `Figlio` derivi dalle classi `Madre` e `Padre`, a loro volta derivate dalla classe `Persona`. Se chiamassimo una funzione della classe `Persona` da un oggetto di classe `Figlio`, quale verrebbe chiamata, quella che ha ereditato da `Padre` o quella che ha ereditato da `Madre`?

```
/**
 * @file ereditarieta-classi-base-virtuali.cpp
 * Gestione delle classi base virtuali.
 */

#include <iostream>
#include <string>

using namespace std;

class Persona {
public:
    void getClass(){
        cout << "Persona" << endl;
    }
};

class Padre : public virtual Persona {
};

class Madre : virtual public Persona {
};

class Figlio : public Padre, public Madre {
};

int main(int argc, char** argv)
{
    Figlio caino;
```

```
    caino.getClass();
    return 0;
}
```

In realtà, nessuna delle due, perché questo codice genera un errore:

```
src/cpp/ereditarieta-classi-base-virtuali.cpp:30:11:
error: non-static member 'getClass' found in multiple base-class
      subobjects of type 'Persona':
    class Figlio -> class Padre -> class Persona
    class Figlio -> class Madre -> class Persona
    caino.getClass();
          ^
src/cpp/ereditarieta-classi-base-virtuali.cpp:13:10:
note: member found by ambiguous name lookup
    void getClass(){
         ^
1 error generated.
```

Puoi evitare questo genere di problemi dichiarando la classe `Persona` come classe base *virtuale* delle classi `Madre` e `Padre`:

```
class Padre : public virtual Persona {
};

class Madre : virtual public Persona {
};
```

In questo modo, la classe `Figlio` erediterà tutti i membri propri delle classi `Madre` e `Padre`, ma solo una copia dei metodi e degli attributi della classe virtuale `Persona` che entrambi contengono.

Come ti ho detto, definire una nuova classe equivale a definire un nuovo tipo di dato, che sarà considerato dal compilatore alla stessa stregua dei dati primitivi del linguaggio. Questo vuol dire, per esempio, che se vogliamo possiamo creare un array di oggetti di classe `Pesce` così come creeremmo un array di `int` o di `char`:

```
/** Array di oggetti di classe Pesce */
Pesce acquario[10];
```

C'è solo una limitazione: siccome tutti gli elementi di un array devono essere inizializzati al momento della sua creazione, la classe deve avere un costruttore di default. Se decidessimo di creare un array di oggetti della classe `Punto` che abbiamo visto nella scorsa lezione, gli elementi dell'array dovranno essere inizializzati esplicitamente:

```
Punto spline[3] = { Punto(3,5), Punto(0,0), Punto(7,7) } ;
```

Possiamo aggiungere a un array di oggetti di una classe base anche degli oggetti appartenenti alle sue classi derivate:

```
Persona famiglia[3] = { Padre(), Madre(), Figlio() } ;
```

Al contrario, gli oggetti della classe base non possono comparire in array di oggetti della classe derivata e un'istruzione come quella qui sotto darà errore:

```
Figlio classe[4] = { Figlio(), Figlio(), Persona(), Figlio() } ;
```

Così come, negli scacchi, la regina può muovere come una torre, ma una torre non può muoversi come una regina, un oggetto di tipo `Persona` non contiene tutta l'informazione relativa a un oggetto di tipo `Figlio` e quindi non può essere usato in sua sostituzione.
Lo stesso discorso fatto per gli array, vale anche per i puntatori. A un puntatore a oggetti di tipo `Persona` può essere assegnato un oggetto di tipo `Figlio`, mentre l'operazione inversa causerà un errore di compilazione:

```
Persona *ptrP = new Figlio() ;  // OK
Figlio  *ptrF = new Persona();  // ERRORE!
```

Il compilatore è in grado di capire la relazione che c'è fra una classe derivata e la sua classe base e può quindi stabilire un cammino di coercizione dal tipo dell'oggetto a quello del puntatore, ma non ha modo di accedere ai membri o alle funzioni di una classe derivata da un oggetto di classe base.
Abbiamo detto a suo tempo che i puntatori sono come delle maschere che isolano determinate sequenze di bit, la cui dimensione varia a seconda del tipo del puntatore. Lo stesso discorso vale anche per le classi: un puntatore di classe base associato a un oggetto di classe derivata "vedrà" solo i dati e le funzioni della sua classe:

```
/**
 * @file ereditarieta-puntatori.cpp
 * Gestione dei puntatori a classi derivate.
 */

#include <iostream>
#include <string>

using namespace std;

class Persona {
public:
    void getClass(){
        cout << "Persona" << endl;
    }
};

class Madre : virtual public Persona {
```

```
public:
    void getClass(){
        cout << "Madre" << endl;
    }
};

int main(int argc, char** argv)
{
    Madre   * ptrM = new Madre;
    Persona * ptrP = ptrM ;
    ptrM->getClass() ;
    ptrP->getClass() ;
    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/ereditarieta-puntatori.cpp -o src/out/esempio
> src/out/esempio
Madre
Persona
```

Non c'è nessun errore: la funzione `getClass()` che interviene nella seconda istruzione di output non è, come ci si aspettava, quella della classe `Madre`, a cui l'oggetto appartiene, bensì quella della classe base, che è l'unica a cui il programma può accedere tramite un puntatore a oggetti di tipo `Persona`.
Questo comportamento (corretto) del programma diventa particolarmente rischioso se la classe ha un distruttore:

```
/**
 * @file ereditarieta-distruttori.cpp
 * Gestione dei distruttori nelle classi derivate.
 */

#include <iostream>
#include <string>

using namespace std;

/** Definiamo una classe base */
class Padre {
public:
    ~Padre() ;
} ;

/** Definiamo delle classi derivate. */
class Figlio : public Padre {
public:
```

```
    ~Figlio() ;
};

class Nipote : public Figlio {
public:
    ~Nipote() ;
};

int main(int argc, char** argv)
{
    /** Creiamo un array di puntatori di classe base. */
    Padre * dinastia[3] ;

    /** Assegnamo delle istanze delle tre classi ai puntatori */
    dinastia[0]= new Padre  ;
    dinastia[1]= new Figlio ;
    dinastia[2]= new Nipote ;

    /** Facciamo qualcosa con i nostri oggetti.. */

    /** ..e poi li eliminiamo */
    delete dinastia[0] ;
    delete dinastia[1] ;
    delete dinastia[2] ;
}
```

Nessun compilatore ti darà mai errore per questo codice, ma il distruttore chiamato, in tutti e tre i casi, sarà quello della classe base `Padre`, con conseguenze che spaziano dal problematico al disastroso.

Per far sì che una funzione membro di una classe derivata possa essere richiamata anche da puntatori a classi base, la si deve dichiarare come `virtual`.

```
/**
 * @file ereditarieta-funzioni-virtuali.cpp
 * Gestione delle funzioni virtuali.
 */

#include <iostream>

using namespace std;

class Persona {
public:
    virtual void getClass(){
        cout << "Persona" << endl;
```

```
    }
};

class Madre : virtual public Persona {
public:
    void getClass(){
        cout << "Madre" << endl;
    }
};

int main(int argc, char** argv)
{
    Madre   * ptrM = new Madre;
    Persona * ptrP = ptrM ;
    ptrM->getClass() ;
    ptrP->getClass() ;
    return 0;
}
```

L'output di questo codice è:

```
> g++ src/cpp/ereditarieta-funzioni-virtuali.cpp \
    -o src/out/esempio
> src/out/esempio
Madre
Madre
```

Le *funzioni virtuali* sono delle funzioni che vengono richiamate in base alla classe dell'oggetto cui appartengono, indipendentemente dal tipo del riferimento o del puntatore che si utilizza. Ciò è reso possibile da un meccanismo chiamato *binding dinamico* o *late binding*, che consiste nel posticipare il *linking* delle funzioni al momento dell'esecuzione del programma, contrariamente a quanto avviene per le funzioni membro normali, che sono collegate al codice in fase di compilazione — il cosiddetto *early binding*.
In pratica, la cosa funziona così: gli indirizzi di tutte le funzioni dichiarate come `virtual` vengono memorizzati in una tabella interna e solo quando una di queste funzioni viene richiamata dal programma, il sistema ne cerca l'indirizzo, effettuandone poi il *linking* in tempo reale. Capisci da te che l'utilizzo delle funzioni virtuali, oltre a comportare un leggero ritardo nel tempo di esecuzione del programma, visto che l'indirizzo della funzione va ben cercato, impegna anche parte delle risorse del sistema per la memorizzazione della tabella degli indirizzi, quindi, come per tutte le cose, è bene non abusarne.
Le regole che riguardano l'utilizzo delle funzioni virtuali sono:

- le versioni delle funzioni delle classi derivate debbono avere il medesimo tipo di ritorno e gli stessi parametri della versione della classe base: se non è così, il compilatore considera differenti le due funzioni e l'effetto "virtuale" si perde;

- una funzione `virtual` non può essere anche `static`: il concetto stesso di funzione virtuale prevede un collegamento fra un oggetto e una funzione; le funzioni statiche sono indipendenti dagli oggetti della loro classe, quindi le due cose sono incompatibili;
- una funzione può essere dichiarata `virtual` solo nella classe base: non è possibile effettuare la dichiarazione in una classe derivata;
- si può ripetere la specifica `virtual` anche nelle classi derivate, ma non è necessario: lo vedi nell'esempio, dove la seconda versione della funzione `getClass()` non ha la parola chiave `virtual` davanti;
- l'utilizzo dell'operatore di risoluzione della portata annulla inevitabilmente l'effetto delle funzioni virtuali.

È possibile dichiarare una funzione virtuale nella classe base senza definirne il comportamento, se si utilizza la sintassi:

```
virtual <tipo> nomefunzione([argomenti]) = 0 ;
```

Questo tipo di funzioni si chiamano *funzioni virtuali pure* e rendono la classe a cui appartengono una *classe astratta*.
Le classi astratte sono delle classi generiche che possono essere utilizzate come capostipiti per una discendenza di classi specializzate, ma che non possono essere utilizzate direttamente. Le regole che si applicano alle classi astratte sono:

- viene considerata astratta qualunque classe che abbia almeno una funzione virtuale pura;
- le funzioni virtuali pure sono ereditate come dalle classi derivate come funzioni virtuali pure, quindi, se una classe derivata non ridefinisce una funzione virtuale pura della sua classe base sarà considerata dal compilatore come una classe astratta;
- non si possono utilizzare classi astratte come argomenti o come tipi di ritorno di funzioni;
- le classi astratte non possono essere il tipo di un oggetto o di una conversione esplicita.

Data una classe astratta `Mammifero`, le istruzioni seguenti causerebbero degli errori di compilazione:

```
void funz(Mammifero m);  // non possono essere argomenti..
Mammifero funz();        // né valori di ritorno..
punt = (Mammifero*)ptr;  // né il tipo di una conversione.
Mammifero pollo;         // né il tipo di un oggetto
```

È possibile, però, dichiarare un puntatore o una *reference* a una classe astratta e utilizzarli per creare degli array o delle code che possano essere utilizzati con istanze di classi diverse:

```
/**
```

```
 * @file ereditarieta-classi-astratte.cpp
 * Gestione delle classi astratte.
 */

#include <iostream>
#include <string>

using namespace std;

enum Sesso { maschio, femmina, boh};
const char* Sessi[] = {"maschio", "femmina", "boh" };

class Mammifero
{
private:

    Sesso _sesso;

public:

    Mammifero(Sesso sesso = boh)
    : _sesso(sesso) {
    }

    virtual void getSesso() {
        cout << Sessi[_sesso] << endl ;
    }

    /** Funzione virtuale pura */
    virtual void getSpecie() = 0;
} ;

class Cane : public Mammifero
{
private:

    string _nome;

public:

    Cane(const char * nome, Sesso sesso)
    : Mammifero(sesso), _nome(nome){
    }

    void getNome() {
        cout << _nome << endl ;
```

```
    }

    /** Definizione della funzione virtuale */
    virtual void getSpecie() {
        cout << "cane" << endl ;
    }
};

int main()
{
    Cane mioCane("Scylla", femmina) ;
    Mammifero& cane = mioCane;

    /**
     * Accede alle funzioni della classe Cane
     * dall'istanza della classe.
     */
    mioCane.getSpecie();
    mioCane.getSesso();
    mioCane.getNome();

    /**
     * Accede alle funzioni della classe Cane
     * dal puntatore alla classe Mammifero.
     */
    cane.getSpecie();
    cane.getSesso();

    return 0;
}
```

Se compili ed esegui questo codice, otterrai:

```
> g++ src/cpp/ereditarieta-classi-astratte.cpp -o src/out/esempio
> src/out/esempio
cane
femmina
Scylla
cane
femmina
```

La classe base `Mammifero` definisce solo un'astrazione, lasciando alle sue classi derivate il compito di definire attributi e metodi specifici per ciascuna specie particolare. Allo stesso modo, la funzione `getSpecie` definisce solo un concetto, non un algoritmo; saranno le singole classi derivate a ridefinire il comportamento della funzione, adattandolo alle proprie esigenze.
È possibile, comunque, definire un comportamento anche per le funzioni virtuali

pure; per la classe `Mammifero` potrebbe essere qualcosa di simile:

```
inline void Mammifero::getSpecie()
{
    cout << "nessuna" << endl ;
}
```

Non potendo esistere oggetti di classe `Mammifero`, però, la versione base della funzione `getSpecie` potrebbe essere richiamata solo facendo uso dell'operatore `::`

```
mioCane.Mammifero::getSpecie();
cane.Mammifero::getSpecie();
```

Buona parte degli esempii e delle cose che ti ho detto in questa lezione le ho prese dal manuale di programmazione in C++ che il Maestro Canaro scrisse nel Secolo scorso, modificandoli per adattarli a questo contesto.
L'esempio originale delle funzioni virtuali, per esempio, era così:

```
//////////////////////////////////////////////////////////////
//
//   CHISONOV.CPP - Utilizzo delle funzioni virtuali
//
//////////////////////////////////////////////////////////////
#include <iostream.h>
//////////////////////////////////////////////////////////////
class A
{
 public:
    A() {} ;
    virtual void ChiSono()
        { cout << "Sono un oggetto di classe A \n" ; }
} ;
//////////////////////////////////////////////////////////////
class B :public A
{
 public:
    B() : A() {} ;
    void ChiSono()
        { cout << "Sono un oggetto di classe B \n" ; }
} ;
//////////////////////////////////////////////////////////////
void main()
{
 B * ptrB = new B ;
 A * ptrA = ptrB ;
```

```
    ptrB->ChiSono() ;
    ptrA->ChiSono() ;
} ;
//////////////////////////////////////////////////////////////
```

Essendo un codice scritto per l'ambiente Microsoft del 1995 , se provassi a compilarlo adesso, con il compilatore GNU, otterresti una lunga serie di errori:

```
> g++ src/cpp/ereditarieta-codice-canaro.cpp -o src/out/esempio
src/cpp/ereditarieta-codice-canaro.cpp:8:10:
fatal error: 'iostream.h' file not found
#include <iostream.h>
         ^~~~~~~~~~~~
src/cpp/ereditarieta-codice-canaro.cpp:13:8:
error: use of undeclared identifier 'cout';
did you mean 'std::cout'?
        { cout << "Sono un oggetto di classe A \n" ; }
          ^~~~
          std::cout
/Library/Developer/CommandLineTools/usr/bin/
../include/c++/v1/iostream:54:33: note:
'std::cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                                ^
src/cpp/ereditarieta-codice-canaro.cpp:21:8:
error: use of undeclared identifier 'cout'

did you mean 'std::cout'?
        { cout << "Sono un oggetto di classe B \n" ; }
          ^~~~
          std::cout
/Library/Developer/CommandLineTools/usr/bin/
../include/c++/v1/iostream:54:33: note: '
std::cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                                ^
src/cpp/ereditarieta-codice-canaro.cpp:24:1:
error: 'main' must return 'int'
void main()
^~~~
```

Il valore didattico di questo codice, però, è immutato. Il *Karma* dei due oggetti è determinato dalla dichiarazione delle loro classi, che non gli lascia altra possibilità che fare ciò per cui sono stati creati. L'output delle funzioni, lo stile dei commenti o il fatto che in un caso le classi si chiamino `A` e `B`, mentre nell'altro si chiamano `Persona` e `Madre`, sono solo differenze formali che non influiscono sul *Dharma* dell'esempio, che è quello di illustrare il comportamento delle funzioni

virtuali. Se in vece del nuovo codice io avessi usato quello originale del Maestro Canaro, tu avresti capito ugualmente; forse anche meglio, perché il nuovo codice sembra migliore a me, ma non è detto che lo sia anche per te.
Lo stesso principio vale anche per l'Universo. Così come gli oggetti all'interno di uno stesso programma occupano posizioni diverse in memoria, a seconda del momento in cui il programma viene eseguito, gli esseri senzienti possono manifestarsi in luoghi e tempi differenti nei diversi cicli di esistenza.
Ciascuno di noi è un orchestrale a cui è stata assegnata una partitura. Possiamo suonarla più o meno bene o non suonarla affatto, nascondendoci nel pieno d'orchestra, ma il nostro valore è solo — permettimi il gioco di parole — strumentale, perché ciò che conta non siamo noi: è la musica; e questa, non è né la prima né l'ultima volta che la suoniamo.
Io, qui, ora, con il mio naso la mia bocca e i miei capelli, ti sto insegnando ciò che so del C'hi++ e tu, che hai il tuo naso la tua bocca e i tuoi capelli, lo stai scrivendo nel tuo libro, ma la stessa informazione che stiamo trasmettendo e perpetuando la potrebbero trasmettere e perpetuare anche persone diverse in un altro tempo o in un altro Pianeta in un altro ciclo dell'Universo. Molto probabilmente abbiamo già avuto questa conversazione in passato e la faremo ancora in futuro. Forse non useremo le stesse parole; forse avremo nomi differenti, forse tu sarai il maestro e io l'allievo, ma la nostra amicizia sarà la stessa, perché quella fa parte della dichiarazione della nostra classe; non può e non deve mutare.

> Le figure mitiche, è vero, nascono e trapassano, ma non proprio come noi mortali. Hanno bisogno di denominazioni caratteristiche, come quella di «Re nel Passato e nel Futuro». Sono esistite in passato? Allora sono esistite ancor prima, o esisteranno ancora, con altri nomi, sotto altri aspetti, proprio come il cielo ci riporta in eterno le sue configurazioni. Se si cercasse di definirle con precisione come persone e cose, sicuramente svanirebbero ai nostri occhi, quanto i frutti di una fantasia malata. Ma se si rispetta la loro vera natura, riveleranno questa natura come funzioni.

Ogni epoca ha i suoi eroi e i suoi demoni; la memoria delle loro battaglie, genera il mito.
La Scienza è transeunte: abbiamo poche notizie riguardo le conoscenze scientifiche dei popoli del passato, mentre conosciamo bene i loro miti, perché il mito è immortale; la Scienza no, a meno che non sia assorbita dal mito e trasformata in leggenda o superstizione. Il Maestro Canaro, per esempio, era convinto che la superstizione relativa ai numeri 13 e 17 fosse nata dall'osservazione del comportamento delle locuste, che, a seconda della specie, passano o tredici o diciassette anni sotto terra in uno stadio larvale , poi escono fuori tutte insieme e spendono la loro breve vita devastando le coltivazioni. La paura dei numeri 13 e 17, secondo lui, era una conoscenza scientifica tramandata nel tempo e nello spazio come superstizione, dalle culture contadine che, ciclicamente, vedevano devastati loro raccolti.

L'arte è il motore del mito. Un motore che si auto-alimenta, come il Sole, perché si nutre di eventi epici e li genera a sua volta ispirando gli eroi a imprese degne di memoria.
La parola *arte*, così come: *amore*, del resto, è una di quelle parole che le gente utilizza spesso ma di cui non viene mai data una definizione precisa, perché le si ritiene dei concetti auto-esplicativi che non occorre definire. È sbagliato: come abbiamo visto, tutte le parole, anche quelle più comuni, possono essere interpretate in maniera differente. In un suo saggio giovanile su Amore e Arte, il Maestro Canaro scrisse che:

> L'Arte è la traccia del cammino dell'Uomo verso Dio

specificando poi che, con il termine: "Dio" (altra parola interpretata in maniera differente da ciascuno di noi), intendeva il senso dell'Esistenza. Alcuni anni dopo, però, guardando delle foto di crostate realizzate dallo chef Gianluca Fusto, capì che la sua definizione era imperfetta, perché non includeva, o quanto meno lasciava a margine, gli arte-fatti che non ricadevano nelle categorie artistiche canoniche. Modificò per ciò la sua definizione di Arte in:

> L'Arte è la traccia del cammino dell'Uomo verso la Perfezione

Non si trattò di una contraddizione, ma di una precisazione, dato che per lui — così come per noi del resto — la ricerca della perfezione era, effettivamente, il senso dell'Esistenza e, quindi, Dio.
Questo episodio della vita del Maestro Canaro ha la peculiarità di dimostrare i principii stessi che afferma: il primo è che non dobbiamo avere paura di mettere in discussione le nostre idee, se ci accorgiamo che sono sbagliate o incomplete; il secondo è che, se affrontiamo la vita nel modo giusto, tutto ciò che facciamo sarà Arte, non solo la disposizione dei fiori o la cerimonia del Té.
Nella vita non esistono momenti di serie *A*, in cui facciamo le cose che ci piacciono e momenti di serie *B*, in cui facciamo ciò che è necessario fare: ogni istante è importante. Per sottolineare questo precetto, il Maestro Canaro definì un'estetica per la disposizione del bucato sullo stendi-panni e, per non correre il rischio di essere preso troppo sul serio, la chiamò *Ikebarba*.
I principii dell'Ikebarba, così come li formulò il Maestro Canaro, sono:

> **L'Ikebarba è fatta per l'uomo, non l'uomo per l'Ikebarba.**
> L'Ikebarba non deve essere un peso per chi la pratica, ma un obbligo giojoso. I panni devono comunque essere messi ad asciugare; il tempo necessario a farlo in maniera sciatta o consapevole è pressocché lo stesso, ma un'Ikebarba ben fatta provvederà panni asciutti in minor tempo e renderà la vista dello stendipanni meno fastidiosa.
>
> **L'Ikebarba comincia nel negozio.**
> Gli indumenti di colore diverso o con colori sgargianti sono difficili da accostare cromaticamente; è preferibile quindi acquistare abiti dalle tinte sobrie e possibilmente intonati gli uni agli altri, in modo

da renderne più facile e più elegante la composizione sullo stendipanni. Attenzione, però: un guardaroba di tipo militare o maoista, con indumenti identici e dello stesso colore è un eccesso da rifuggire, perché renderebbe monotona la composizione (e non solo quella).

L'Ikebarba rifugge le mollette. Le mollette sono utili come la psicanalisi: è l'equilibrio che deve tenere i panni sui fili, non una forza di coercizione esterna. I diversi capi devono essere posti sul filo in modo che il peso di una parte bilanci quello dell'altra.

Esistono tre tipi di Ikebarba:

- **cromatica:** quando i panni vengono posizionati sullo stendino in base al loro colore;
- **funzionale:** quando gli indumenti sono posizionati in funzione dei rispettivi tempi di asciugatura, ponendo i capi pesanti all'esterno, dove ricevono più aria, e quelli più leggeri o sintetici all'interno;
- **perfetta:** quando gli aspetti estetici e funzionali si fondono in un tutt'uno armonico.

Come puoi facilmente intuire, gli intenti del Maestro Canaro erano per buona parte ironici (mi confessò che la prima formulazione della disciplina era nata come un tentativo di dissimulare la sua ossessione per l'ordine) e per lungo tempo fu indeciso se includerla o meno nel corpo della Dottrina. Si decise a farlo quando capì che le sue perplessità nascevano proprio da quelle forme di prevenzione che l'Ikebarba doveva contrastare. Come ci insegnano Banzan e Paul Simon, la Verità è ovunque, se la sappiamo cercare, anche sui muri delle metropolitane o nelle botteghe dei mercati.
A ogni modo, l'Ikebarba può davvero comportare dei benefici per chi la pratica. In primo luogo, modera l'effetto nefasto dell'Annosa Dicotomia e dei suoi servitori *Marketing* e *Moda*, che ci spingono ad acquistare indumenti che non ci occorrono e che sfrutteremo solo per breve tempo. Riducendo le variazioni cromatiche del bucato, poi, riduce anche il numero di lavaggi settimanali e con esso il fabbisogno di energia elettrica, acqua, prodotti detergenti e plastica. Anche il ripudio delle mollette ha una sua valenza funzionale: se i capi sono messi ad asciugare a cavallo dei fili, l'acqua nel tessuto tenderà a scendere da entrambi i lati, riducendo il tempo dell'asciugatura.
Indubbiamente, il fatto che la Regola del nostro Ordine ci imponga l'uso di camicie bianche non risolve il problema dell'inquinamento, ma, come diceva il Maestro Canaro:

> La pelliccia è fatta di peli

e finché la nostra specie non imparerà a fare un uso più responsabile delle sue gonadi, non potremo far altro che compensare come possiamo i problemi legati alla sovrappopolazione.

Il polimorfismo

> When Me they fly, I am the wings
> I am the double and the int

Come avrai certamente intuito da tutto ciò che abbiamo detto finora, la caratteristica principale del C++ è il polimorfismo.

Avevamo iniziato a parlarne durante la lezione introduttiva sul C++ e l'avevamo illustrato con un esempio che, a questo punto, non dovrebbe più avere segreti, per te:

```
/**
 *  @file src/cplusplus-template.cpp
 *  Esempio utlizzo dei template di classi.
 */

#include <iostream>
#include <ctime>
#include <cstring>
#include <list>

using namespace std;

/** Definisce due nuovi tipi di dato */
typedef time_t Data;
typedef enum _sesso {
    maschio = 'm',
    femmina = 'f'
} Sesso;

/** Definisce una classe base per la gestione degli animali */
class Animale {
private:
    string _razza;
    Sesso  _sesso;
public:
    /** Costruttori di copia e parametrico */
    Animale() {}
    Animale(const char* razza, const Sesso sesso ) {
        _razza  = razza;
        _sesso  = sesso;
    }
    /** Funzione virtuale pura: rende la classe "astratta" */
```

```
    virtual const char* getSpecie() const {
        return "";
    }
    /** Funzioni di interfaccia */
    const char getSesso() const {
        return (char)_sesso;
    }
    const char* getRazza() const {
        return _razza.c_str();
    }
};

/** Operatore di output su stream per la classe Animale */
ostream& operator << (ostream& os, const Animale& animale) {
    os  << "Specie:" << animale.getSpecie() << "\t"
        << "Razza:"  << animale.getRazza()  << "\t"
        << "Sesso:"  << animale.getSesso()
        << endl;
    return os;
}

/** Definizione della classe derivata Cavallo */
class Cavallo : public Animale {
public:
    /** Definizione dei costruttori della classe */
    Cavallo() {}
    Cavallo(const char* razza, const Sesso sesso )
    : Animale(razza, sesso ) {
    }
    /** Ridefinizione della funzione virtuale pura */
    const char* getSpecie() const {
        return "Cavallo";
    }
};

/** Definizione della classe derivata Cavallo */
class Asino : public Animale {
public:
    /** Definizione del costruttore della classe */
    Asino() {}
    Asino(const char* razza, const Sesso sesso )
    : Animale(razza, sesso ) {
    }
    /** Ridefinizione della funzione virtuale pura */
    const char* getSpecie() const {
        return "Asino";
```

```
    }
};

/** Definizione della classe Monta */
class Monta {
private:
    Animale* _maschio;
    Animale* _femmina;
    Data     _giorno;
    string   _esito;
    /**
    *   Funzione privata per la definizione
    *   dell'esito della monta
    */
    void setEsito() {
        if(strcmp(_maschio->getSpecie(),"Asino") == 0) {
            if(strcmp(_femmina->getSpecie(),"Asino") == 0) {
                _esito = "asino";
            } else {
                _esito = "mulo";
            }
        } else {
            if(strcmp(_femmina->getSpecie(),"Cavallo") == 0) {
                _esito = "puledro";
            } else {
                _esito = "bardotto";
            }
        }
    }
public:
    /** Costruttore della classe */
    Monta(Animale* maschio, Animale* femmina) {
        _maschio = maschio;
        _femmina = femmina;
        time(&_giorno);
        setEsito();
    }
    /** Operatore di output, "friend" della classe Monta */
    friend ostream& operator<<(ostream& os, const Monta& copula) {
        os << "DATA:    " << asctime(localtime(&copula._giorno))
           << "MASCHIO: " << *copula._maschio
           << "FEMMINA: " << *copula._femmina
           << "ESITO:   " << copula._esito
           << endl;
           return os;
     };
```

```
};

int main()
{
    /**
     *  Crea gli oggetti di classe derivata
     *  e li assegna a puntatori della classe base.
     */
    Animale* cavallo  = new Cavallo("lipizzano", maschio);
    Animale* giumenta = new Cavallo("maremmano", femmina);
    Animale* asino    = new Asino("amiatino", maschio);
    Animale* asina    = new Asino("sardo", femmina);

    /** Crea una lista con una classe template */
    list<Monta> monte;

    /** Associa alla lista degli oggetti di classe Monta */
    monte.push_back(Monta(cavallo, giumenta));
    monte.push_back(Monta (asino, asina));
    monte.push_back(Monta (asino, giumenta));
    monte.push_back(Monta (cavallo, asina));

    /** Mostra il contenuto della lista */
    list<Monta>::iterator it;
    for (it=monte.begin(); it!=monte.end(); it++) {
        cout << *it << endl;
    }

    return 0;
}
```

L'output di questo codice, nel caso l'avessi scordato, è:

```
> g++ src/cpp/cplusplus-template.cpp -o src/out/esempio
> src/out/esempio
DATA:    Sun May 23 14:42:51 2021
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f
ESITO:   puledro

DATA:    Sun May 23 14:42:51 2021
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
FEMMINA: Specie:Asino   Razza:sardo     Sesso:f
ESITO:   asino

DATA:    Sun May 23 14:42:51 2021
MASCHIO: Specie:Asino   Razza:amiatino  Sesso:m
```

```
FEMMINA: Specie:Cavallo Razza:maremmano Sesso:f
ESITO:   mulo

DATA:    Sun May 23 14:42:51 2021
MASCHIO: Specie:Cavallo Razza:lipizzano Sesso:m
FEMMINA: Specie:Asino   Razza:sardo     Sesso:f
ESITO:   bardotto
```

Prima di andare avanti, però, è necessario fare un po' di chiarezza su tre termini legati al polimorfismo: *overload*, *override* e *ridefinizione*.
Con il termine: **overload** di una funzione si intende la una funzione che abbia lo stesso nome di un'altra, ma dei parametri differenti. Un tipico esempio di *function overload* sono le differenti versioni del costruttore di una classe:

```
Cavallo() {}
Cavallo(const char* razza, const Sesso sesso )
: Animale(razza, sesso ) {
}
```

Le due funzioni hanno lo stesso nome e il compilatore sceglierà l'una o l'altra in base ai parametri che vengono utilizzati.
Una funzione **overridden** è una funzione che ha una definizione diversa da quella di una funzione virtuale di una sua classe-base:

```
const char* getSpecie() const {
    return "Asino";
}
```

Come abbiamo visto, il compilatore sceglie l'una o l'altra in base al tipo di oggetto utilizzato per la chiamata.
Se la funzione della classe base non fosse stata virtuale, questa sarebbe stata una semplice **ridefinizione**:

```
class Persona {
public:
    void getClass(){
        cout << "Persona" << endl;
    }
};
```

Quando gestisce queste funzioni, il compilatore non fa un controllo di tipo dinamico, basato sul tipo dell'oggetto al momento dell'esecuzione, ma sceglie la funzione da chiamare in base al tipo di puntatore o riferimento utilizzato, cosa che, come sai, può creare dei problemi:

```
Madre   * ptrM = new Madre;
Persona * ptrP = ptrM ;
ptrM->getClass() ;
ptrP->getClass() ;  // chiama la funzione di Persona - ERRORE
```

Alla luce di tutto ciò, possiamo correggere i commenti del codice di esempio:

```
/** Overload dell'operatore di output su stream  */
ostream& operator << (ostream& os, const Animale& animale) {
    os  << "Specie:" << animale.getSpecie() << "\t"
        << "Razza:"  << animale.getRazza()  << "\t"
        << "Sesso:"  << animale.getSesso()
        << endl;
    return os;
}

/** Override della funzione virtuale pura */
const char* getSpecie() const {
    return "Cavallo";
}

/** Override della funzione virtuale pura */
const char* getSpecie() const {
    return "Asino";
}

/** Overload dell'operatore di output su stream */
friend ostream& operator << (ostream& os, const Monta& copula) {
    os << "DATA:    " << asctime(localtime(&copula._giorno))
       << "MASCHIO: " << *copula._maschio
       << "FEMMINA: " << *copula._femmina
       << "ESITO:   " << copula._esito
       << endl;
       return os;
 };
```

Nel C++, a ogni operatore corrisponde una funzione. Quella dell'operatore binario `+=`, per esempio, è:

```
<tipo>& operator +=  (<tipo>& a, <tipo>& b) ;
```

laddove `a` e `b` sono i due oggetti che intervengono nell'operazione e `<tipo>` è il tipo delle variabili che intervengono nell'operazione.

```
int&    operator +=  (int&    a, int&    b) ;
float&  operator +=  (float&  a, float&  b) ;
double& operator +=  (double& a, double& b) ;
```

Dato che gli operatori unari possono essere prefissi o postfissi, per consentire al compilatore di distinguere le funzione corretta da utilizzare, alla funzione dell'operatore postfisso si aggiunge un secondo parametro, non utilizzato:

```
void operator ++ (<tipo> a) ;           // versione prefissa
```

```
void operator ++ (<tipo> a, <tipo>) ;   // versione postfissa
```

Le funzioni degli operatori *overloaded* possono essere richiamate in maniera diretta. Le due istruzioni qui sotto, una volta compilate, producono il medesimo codice e lo stesso risultato. Se riesci a trovare una qualunque ragione per usare la prima sintassi piuttosto che la seconda, fallo pure:

```
a = b.operator + (c) ;
a = b + c ;
```

Il comportamento degli operatori è predefinito per tutti i tipi standard e può essere ridefinito per gestire anche dei tipi di dato aggregati come le strutture o le classi. La classe `string`, della libreria standard del C++, per esempio, ridefinisce, fra le altre cose, il comportamento degli operatori di assegnazione `+=` e `+` e dell'operatore di output su stream `<<` in modo che si possano compiere delle operazioni sulle stringhe con la stessa sintassi che si utilizza per altri tipi di dato:

```
/**
 * @file src/polimorfismo-operatori.cpp
 * Esempio di overload di un operatore.
 */

#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string s1 ("Pip");
    string s2 ("po");
    const char* s3 = "Plut";

    /**
     *  La classe string definisce tre overload
     *  per l'operatore += .
     *
     *  string& operator+= (const string& str);
     *  string& operator+= (const char* s);
     *  string& operator+= (char c);
     */
    s1 += s2;
    s1 += s3;

    cout << (s1 + 'o') << endl;

    return 0;
```

```
}
```

L'output di questo codice è ben noto:

```
> g++ src/cpp/polimorfismo-operatori.cpp -o src/out/esempio
> src/out/esempio
PippoPluto
```

Lo stesso risultato si può ottenere anche con la funzione `append`:

```
string& append (const string& str)
```

ma utilizzare un operatore standard rende il codice più facile da leggere e da scrivere, se non altro perché non ti devi ricordare come si chiama la funzione per unire due stringhe.
Gli unici operatori che non possono essere ridefiniti da una classe sono:

- l'operatore di selezione `.`;
- l'operatore di risoluzione di indirizzamento dei puntatori a membri della classe `.*`;
- l'operatore di risoluzione del campo d'azione `::`;
- l'operatore condizionale `? :`;
- i simboli `#` e `##`, che vengono utilizzati dal preprocessore.

Tranne alcune eccezioni che vedremo fra poco, tutti gli operatori del C++ possono essere ridefiniti o come funzione membro di una classe o come funzione globale:

```
/**
 * @file src/polimorfismo-in-out.cpp
 * Operatori come funzioni membro o globali.
 */

#include <iostream>
#include <string>

using namespace std;

struct A
{
    int _a;
    A(int a) : _a(a) {}
};

struct B
{
    int _b;
    B(int b) : _b(b) {};
```

```
    /** Overload come funzione membro */
    int operator + (const A& a) {
        return _b + a._a;
    }
};

/** Overload come funzione globale */
int operator + (const A& a, const B& b) { return a._a + b._b; }

int main ()
{
    struct A a(3);
    struct B b(5);

    cout << (a + (b + a)) << endl;

    return 0;
}
```

Quando si ridefinisce il comportamento di un operatore per una classe, bisogna tenere conto della visibilità dei dati membro che deve utilizzare. Se l'operatore, com'è probabile, deve gestire dei dati privati o protetti, le possibilità sono due: o sfruttare le funzioni di interfaccia della classe o dichiarare l'operatore `friend` della classe. Nell'esempio iniziale sono applicate entrambe le possibilità: l'operatore di output su stream per la classe `Animale` utilizza le funzioni di interfaccia della classe:

```
ostream& operator << (ostream& os, const Animale& animale) {
    os  << "Specie:" << animale.getSpecie() << "\t"
        << "Razza:"  << animale.getRazza()  << "\t"
        << "Sesso:"  << animale.getSesso()
        << endl;
    return os;
}
```

mentre l'operatore di output per la classe `Monta` è dichiarato come `friend` della classe e quindi può accedere direttamente ai dati membro dell'istanza:

```
friend ostream& operator << (ostream& os, const Monta& copula) {
    os << "DATA: "    << asctime(localtime(&copula._giorno))
       << "MASCHIO: " << *copula._maschio
       << "FEMMINA: " << *copula._femmina;
       return os;
};
```

La scelta fra l'una o l'altra possibilità dipende dal tipo di programma che devi scrivere: se punti alla velocità, scegli la seconda, che è più diretta, altrimenti scegli la prima, che sarà probabilmente più lenta in esecuzione, ma non necessi-

terà di riscritture in caso di modifiche alla struttura della classe. Non è possibile, però, ridefinire come funzione membro di una classe una funzione operatore che abbia come primo parametro una classe di cui non si ha il controllo (come, per esempio, la funzione operatore `<<` che ha come primo parametro un riferimento a `ostream`) perché nella funzione membro questo parametro sarebbe sostituito dal parametro implicito `this`, che ha un altro tipo di dato, causando un errore di compilazione.
Gli operatori `=`, `()`, `[]` e `->` non possono essere ridefiniti come funzioni globali, ma devono sempre essere implementati come funzione membro non statica di una classe Le altre regole da ricordare, in questi casi, sono:

- l'operatore unario di assegnamento `=` è l'unico caso di funzione membro che non viene ereditata da eventuali classi figlie; se non viene ridefinito, prevede l'assegnamento membro a membro degli attributi e ha la sintassi:

  ```
  C& C::operator = (const C& origine) ;
  ```

- l'operatore binario `[]` permette di implementare vettori di tipo particolare, mantenendo una sintassi standard e ha la forma:

  ```
  c.operator [] (n) ;
  ```

  dove `c` è un oggetto di classe `C` e l'indice n può essere un qualsiasi tipo di dato ;

- per ridefinire l'operatore binario di chiamata a funzione `()`, va utilizzata la sintassi:

  ```
  c.operator()(p) ;
  ```

  dove `c` è sempre un oggetto di classe `C` e `p` è un elenco anche vuoto, di parametri;

- l'operatore unario di accesso ai membri della classe `->` viene interpretato come:

  ```
  (C.operator -> ())->m ;
  ```

  e ritorna o un oggetto o un puntatore a un oggetto di classe `C`.

Ridefinire gli operatori `new` e `delete`, il cui comportamento è strettamente legato all'hardware, potrebbe non essere una scelta astuta dal punto di vista della portabilità del codice; detto ciò, se una classe ha bisogno di gestire la memoria in modo particolare, lo può fare, ma deve rispettare due regole:

- l'operatore `new` deve avere il primo argomento di tipo `size_t` e restituire un puntatore a `void`;
- l'operatore `delete` deve essere una funzione di tipo `void` che abbia un primo argomento di tipo `void*` e un secondo argomento, facoltativo, di tipo `size_t`.

In C, per trasformare un `int` in un `double` si utilizzano gli operatori di cast:

```
long int i = 5 ;
double d = (double) i ;
```

Il C++ accetta questa sintassi, così come accetta che si usi `malloc` al posto di `new`, ma la sua sintassi standard, che ricorda vagamente i costruttori delle classi, prevede che il dato da convertire sia passato come parametro a una funzione con lo stesso nome del tipo in cui si vuole che avvenga la conversione :

```
long int i = 5 ;
double d = double(i) ;
```

Il compilatore del C++ ha la possibilità di convertire un qualunque tipo di dato primitivo in un altro, ma non può sapere come comportarsi con i tipi di dato definiti dall'utente; dobbiamo quindi istruirlo, così come abbiamo fatto con i costruttori e gli operatori, definendo dei cammini di coercizione dai tipi di dato primitivi e viceversa. Il primo caso, ovvero la trasformazione dal tipo primitivo a quello definito dall'utente, è il più semplice: di fatto si tratta di definire, laddove non ci sia già, un costruttore per la nuova classe che richieda dei parametri di tipo primitivo. Quando invece non esiste un costruttore da estendere, ovvero quando la coercizione è dal tipo definito dall'utente a un tipo di dato primitivo o fornito in una libreria di cui non si possiede il codice sorgente, è necessario ridefinire l'operatore di conversione `()`.
Immagina di aver creato un nuovo tipo di dato `Frazione` per la gestione dei numeri razionali. Per poterlo utilizzare in espressioni contenenti dati di tipo primitivo dovresti ridefinire ciascun operatore per fargli accettare dei dati di tipo misto, sia come primo che come secondo parametro:

```
Frazione operator + (int i, Frazione f) :
Frazione operator - (int i, Frazione f) :
Frazione operator + (double i, Frazione f) :
Frazione operator - (double i, Frazione f) :
...
Frazione operator + (Frazione f, int i) :
Frazione operator - (Frazione f, int i) :
Frazione operator + (Frazione f, double i) :
Frazione operator - (Frazione f, double i) :
```

Puoi risparmiarti questa seccatura ridefinendo solo il comportamento degli operatori per la nuova classe e fornendo al compilatore dei cammini di conversione dai tipi primitivi al nuovo tipo di dato, in modo che possa trasformare i dati nel tipo appropriato, nel caso di espressioni miste:

```
/**
 * @file src/polimorfismo-cast.cpp
 * Gestione della conversione esplicita.
 */
```

```
#include <iostream>

using namespace std;

class Frazione
{
private:

    int _num ;
    int _den ;

public:

    /** Costruttore con parametri interi */
    Frazione(int n, int d = 1)
    : _num(n), _den(d) {}

    /**
     * Costruttore con parametro a virgola mobile.
     * La definizione è piuttosto complessa, te la risparmio.
     */
    Frazione(double d) ;

    /** Overload dell'operatore di cast a intero */
    operator int () {
        return _num / _den ;
    }

    /** Overload dell'operatore di cast a double */
    operator double() {
        return (double) _num / (double) _den ;
    }

    /** Overload degli operatori di somma e sottrazione */
    friend Frazione operator+ (Frazione f1, Frazione f2);
    friend Frazione operator- (Frazione f1, Frazione f2);

};
```

L'ultima cosa di cui ti devo parlare, a proposito del polimorfismo, sono i *template*.
I template, nel C++, sono dei modelli che si utilizzano per definire delle funzioni o delle classi polivalenti. Se uno stesso compito può essere eseguito in maniera simile su parametri di tipo differente, invece di scrivere delle funzioni o delle classi identiche per ciascun tipo di parametro, si può scrivere una funzione o una

classe template e richiamarla ogni volta con il tipo di parametro appropriato:

```
int    somma(int    a, int    b) { return a + b; }
float  somma(float  a, float  b) { return a + b; }
double somma(double a, double b) { return a + b; }

template <class T>
somma(T a, T b) { return a + b; }
```

Quando il compilatore trova nel codice un template, sia esso la dichiarazione di una classe o una chiamata a funzione, la sostituisce con il codice corrispondente, così come avviene per le macro-istruzioni del precompilatore, ma, a differenza di quello che avviene per le macro, il tipo dei parametri del template è sottoposto a uno stretto controllo, così come il resto del codice.
Il formato per la dichiarazione di una *funzione template* è:

template <class *identificatore*> *dichiarazione*;
template <typename *identificatore*> *dichiarazione*;

Non c'è nessuna differenza fra la prima e la seconda forma: sia `class` che `typename` producono lo stesso effetto.
identificatore è un simbolo che identifica un determinato tipo di dato o una classe definita dall'utente. Per esempio, la sintassi di una funzione template che torna il maggiore di due parametri sarà qualcosa di simile:

```
template<class T>
T maggiore (T x, T y) {
    return (x > y) ? x : y;
}
```

In questo caso, l'identificativo del tipo è la lettera `T` che compare sia fra gli apici nella prima riga che fra parentesi nella seconda, ma può essere qualsiasi stringa.
I parametri possono essere più di uno:

```
template<class C1, class C2>
T funz (C1 x, C2 y) {
...
}
```

e possono avere un valore di default:

```
template<class N = int>
T funz (N n) {
...
}
```

La chiamata delle funzioni template è simile a quella delle funzioni ordinarie, con l'aggiunta del tipo dei parametri che devono essere gestiti:

```
cout << maggiore<int>   (  9,  12) << endl;
cout << maggiore<double>(0.4, 1.2) << endl;
```

```
cout << maggiore<char>  ('a', 'z') << endl;
```

Il prossimo esempio mostra la differenza fra una macro del precompilatore e una funzione template:

```
/**
 * @file src/polimorfismo-macro-template.cpp
 * Funzioni template e macro precompilatore.
 */

#include <iostream>
#include <functional>

using namespace std;

/**
*  Definzione di una macro istruzione per il
*  precompilatore: nessun controllo di tipo.
*/
#define MAGGIORE(a,b) ((a > b) ? a : b)

/**
*  Definzione di una funzione template
*  che torna il maggiore fra due parametri.
*/
template<class T>
T maggiore (T x, T y) {
    return (x > y) ? x : y;
}

int main ()
{
    int    a = 10;
    short  b = 0;
    double d = 3.123456789;

    /** Utilizzo della macro */
    cout << MAGGIORE(9,12)     << endl;
    cout << MAGGIORE(0.4, 0.7) << endl;
    cout << MAGGIORE('a', 'z') << endl;

    /**
    *  La stessa funzione si può utilizzare
    *  con tipi di dato diversi:
    */
    cout << maggiore<int>   (  9,  12) << endl;
    cout << maggiore<double>(0.4, 1.2) << endl;
```

```
    cout << maggiore<char>  ('a', 'z') << endl;

    /** Errore: confronta un carattere con un double */
    cout << MAGGIORE('a', d) << endl;

    /**
    *   Errore: il compilatore non sa quale
    *   dei due tipi di dato utilizzare.
    */
    cout << maggiore(a, b) << endl;

    return 0;
}
```

La macro `MAGGIORE` e la funzione template `maggiore` eseguono la stessa operazione: confrontano i due parametri che hanno ricevuto in input e tornano il maggiore dei due. La grossa differenza fra questi due approcci è che, mentre il tipo dei parametri del template è verificato dal compilatore, la macro è una banale sostituzione che non fa alcun controllo sulle variabili che utilizza. L'istruzione:

```
cout << MAGGIORE('a', b) << endl;
```

compara un carattere con un double e, senza dare problemi in compilazione torna il valore `97`, corrispondente al codice ASCII della lettera `a`. Al contrario, l'istruzione:

```
int   a = 10;
short b = 0;
cout << maggiore(a, b) << endl;
```

causa un errore di compilazione perché i due parametri sono di tipo differente:

```
> g++ src/cpp/polimorfismo-template.cpp -o src/out/esempio
src/cpp/polimorfismo-template.cpp:52:13:
    error: no matching function for call to 'maggiore'
    cout << maggiore(a, b) << endl;
            ^~~~~~~~
src/cpp/polimorfismo-template.cpp:22:3:
    note: candidate template ignored:
        deduced conflicting types for parameter 'T'
      ('int' vs. 'short')
T maggiore (T x, T y) {
  ^
```

La dichiarazione di una *classe template* ha la forma:

template <class *identificatore*> *dichiarazione*;

La lista dei parametri fra i simboli `<>` può contenere uno o più simboli per i

tipi dato gestiti dalla classe. L'utilizzo di queste classi è simile a quello delle funzioni template:

```
/**
 * @file src/polimorfismo-classe-template.cpp
 * Esempio di classe template.
 */

#include <iostream>

using namespace std;

/**
*   Definisce una classe che gestisce coppie
*   di coordinate.
*/
template<class T>
class Coord {
private:
    /** Dati membro con tipo variabile */
    T _x, _y;
public:
    /** Costruttore con tipo di parametri variabile */
    Coord(const T x, const T y)
    : _x(x), _y(y) {
    }
    friend ostream& operator << (ostream& o, const Coord& c) {
        o << c._x << ',' << c._y ;
        return o;
    }
};

int main ()
{
    /** Istanza con coordinate geografiche */
    Coord<double> obelisco(41.903219, 12.458157);

    /** Istanza con coordinate schermo */
    Coord<int>    pixel(821, 134);

    cout << "Obelisco:" << obelisco << endl;
    cout << "Pixel:   " << pixel    << endl;

    return 0;
}
```

Il codice che ti ho mostrato all'inizio di questa lezione utilizza una classe template:

```
list<Monta> monte;
```

La classe `list` è una delle classi della *Standard Template Library* del C++, una libreria di classi e di funzioni che permettono di risolvere dei problemi comuni della programmazione, come la memorizzazione, l'ordinamento o la ricerca di una serie di dati. Le componenti della STL è sono:

- una libreria di **container** che permettono di immagazzinare oggetti e dati;
- degli **iteratori** che consentono di scorrere il contenuto dei container;
- una collezione di **algoritmi** che permettono di eseguire delle operazioni di ordinamento e ricerca su insiemi di dati;
- degli oggetti-funzioni, o: **functors**, che incapsulano una specifica funzione.

La classe `list` è un esempio di container e rappresenta un elenco di elementi memorizzati in aree non contigue della memoria. Al contrario, la classe `vector` implementa un elenco di elementi memorizzati in un'unica area di memoria, così come avviene per gli array del C.
Tutti i vettori della STL posseggono delle funzioni membro che consentono di gestirne gli elementi; la funzione `push_back`, per esempio, aggiunge un elemento in coda alla lista:

```
monte.push_back(Monta(cavallo, giumenta));
monte.push_back(Monta (asino, asina));
monte.push_back(Monta (asino, giumenta));
monte.push_back(Monta (cavallo, asina));
```

Gli *iteratori* sono dei costrutti che permettono di scorrere il contenuto di un container, individuandone gli elementi. Ne abbiamo utilizzato uno nell'istruzione:

```
list<Monta>::iterator it;
for (it=monte.begin(); it!=monte.end(); it++) {
    cout << *it << endl;
}
```

La prima istruzione del ciclo `for` assegna all'iteratore `it` il primo elemento della lista, tornato dalla funzione membro `monte.begin`. La seconda istruzione, verifica che l'iteratore sia differente da `monte.end`, che punta alla fine della lista. La terza istruzione incrementa l'iteratore di una posizione e dimostra come la ridefinizione di un operatore per una classe renda il codice più facile da leggere: anche se tu non hai mai visto una classe template, capisci subito che quella istruzione incrementa il valore di `it` di un'unità.
Gli *algoritmi* della STL, definiti nell'header `<algorithm>` sono funzioni template che permettono di individuare, copiare, ordinare, unire o eliminare i dati all'interno di un container.

```
/**
 * @file src/polimorfismo-algoritmi.cpp
```

```
 * Esempio di utilizzo degli algoritmi della STL.
 */

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    /** Crea il vettore */
    vector<int> vect;

    /** Aggiunge dei valori al vettore */
    vect.push_back(10);
    vect.push_back(70);
    vect.push_back(21);
    vect.push_back(49);
    vect.push_back(35);

    /** Mostra il valore più alto e il più basso */
    cout << *min_element(vect.begin(), vect.end()) << endl;
    cout << *max_element(vect.begin(), vect.end()) << endl;

    /** Mostra i valori nell'ordine in cui sono stati inseriti */
    vector<int>::iterator i;
    for(i = vect.begin(); i != vect.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;

    /** Ordina i valori dal più basso al più alto e li visualizza */
    int n = vect.size();
    sort(vect.begin(), vect.end());
    for (int i=0; i<n; i++) {
        cout << vect[i] << " ";
    }
    cout << endl;

    /** Ordina i valori dal più alto al più basso e li visualizza */
    reverse(vect.begin(), vect.end());
    for (int i=0; i<n; i++) {
        cout << vect[i] << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> g++ src/cpp/polimorfismo-algoritmi.cpp -o src/out/esempio
> src/out/esempio
10
70
10 70 21 49 35
10 21 35 49 70
70 49 35 21 10
```

Le function-class o: *functors* sono delle classi che ridefiniscono il comportamento dell'operatore `()` e che possono quindi agire come se fossero delle funzioni:

```
/**
 * @file src/polimorfismo-functor-stl.cpp
 * Esempio di function objects della STL.
 */

#include <iostream>
#include <functional>

using namespace std;

int main ()
{
    int a = 12;
    int b = 4;

    /** Dichiarazione di oggetti functor */
    plus<int>       p;
    minus<int>      m;
    multiplies<int> x;
    divides<int>    d;
    modulus<int>    o;

    /** Utilizzo degli oggetti come fossero delle funzioni */
    cout << "plus: "       << p(a,b) << endl;
    cout << "minus: "      << m(a,b) << endl;
    cout << "multiplies: " << x(a,b) << endl;
    cout << "divides: "    << d(a,b) << endl;
    cout << "modulus: "    << o(a,b) << endl;

    /** "esegue"" l'oggetto o con nuovi parametri */
```

```
    cout << "modulus: "    << o(a,5) << endl;

    return 0;
}
```

Utilizzati così, i *functor* hanno poco senso, ma possono essere (e sono) molto utili quando si utilizzano quelle funzioni della STl che elaborano tutti gli elementi di un container, come per esempio la funzione `transform`:

```
/**
 * @file src/polimorfismo-rot13.cpp
 * Trasformazione di una stringa con una funzione.
 */

#include <string>
#include <cctype>
#include <iostream>
#include <functional>

using namespace std;

/**
*   Funzione che converte i caratteri di una stringa in rot13
*   Il ROT13 è un algoritmo di cifratura piuttosto banale,
*   perché incrementa di 13 il valore di ciascun carattere.
*   Non è un algoritmo realmente sicuro, però, perché per
*   decifrare il testo crittografato basta crittografarlo
*   di nuovo.
*/
unsigned char rot13(unsigned char c)
{
    unsigned char rot = c;
    if (isalpha(c)) {
        rot = ((tolower(c) - 'a') < 14) ? c + 13 : c - 13;
    }
    return rot;
}

int main ()
{
    string pp("PippoPluto");

    /** Elabora la stringa con la funzione transform */
    transform(
               pp.begin()   // inizio del container da modificare
             , pp.end()     // fine del container da modificare
             , pp.begin()   // container di output
```

```
            , rot13        // funzione da applicare
            );

    cout << pp << endl;

    /** Ripetendo l'operazione, il testo torna normale. */
    transform(
              pp.begin()
            , pp.end()
            , pp.begin()
            , rot13
            );

    cout << pp << endl;

    return 0;
}
```

Se compili ed esegui questo programma, otterrai :

```
> g++ src/cpp/polimorfismo-rot13.cpp -o src/out/esempio
> ./src/out/esempio
CvccbCyhgb
PippoPluto
```

Le funzioni ordinarie ti permettono di sfruttare l'algoritmo `transform` per cifrare un testo con un valore fisso, ma non puoi fare la stessa cosa utilizzando una chiave variabile, perché il quarto parametro non accetta funzioni con più di un parametro. Se provassi a utilizzarlo con qualcosa come:

```
unsigned char cifra(unsigned char c, int chiave)
{
    return c + chiave;
}
```

otterresti l'errore:

```
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/algorithm:1855:34: error: too
      function call, expected 2, have 1
        *__result = __op(*__first);
                    ~~~~         ^
src/cpp/polimorfismo-transform-chiave.cpp:25:5: note: in instantiation of function template
      'std::__1::transform<std::__1::__wrap_iter<char *>, std::__1::__wrap_iter<char *>, uns
      (*)(unsigned char, int)>' requested here
    transform(
    ^
```

È in questi casi che tornano utili i *functor*, perché possono essere inizializzati con uno o più valori specifici e poi essere utilizzati come funzioni unarie:

```
/**
 * @file src/polimorfismo-functor.cpp
 * Creazione di una classe functor.
 */

#include <string>
#include <cctype>
#include <iostream>
#include <functional>

using namespace std;

/** Dichiarazione della classe functor */
class Cifra
{
private:

    int _chiave;

public:

    /**
    *   Il costruttore della classe ha come parametro
    *   il valore della chiave di cifratura
    */
    Cifra(int chiave) : _chiave(chiave) {  }

    /** Ridefinizione dell'operatore () */
    unsigned char operator () (unsigned char c) const {
        return c + _chiave;
    }
};

int main ()
{
    string pp("PippoPluto");

    /**
    *   Richiama transform passando come parametro
    *   un'istanza del functor, inizializzata con
    *   la chiave di cifratura.
    */
    transform(
                pp.begin()
              , pp.end()
              , pp.begin()
```

```
            , Cifra(1)
            );

    cout << pp << endl;

    return 0;
}
```

Compilando ed eseguendo questo programma, ottieni :

```
> g++ src/cpp/polimorfismo-functor.cpp -o src/out/esempio
> ./src/out/esempio
QjqqpQmvup
```

che corrisponde ai caratteri della stringa *PippoPluto* incrementati di un'unità.

Da migliaia di anni, gli uomini cercano di capire quale sia il significato dell'Esistenza.
Le risposte che si sono dati variano a seconda del periodo storico e del territorio in cui il profeta o il filosofo ha vissuto, ma hanno tutte una particolarità: richiedono ai loro seguaci l'accettazione di postulati non dimostrabili, come l'esistenza di una o più divinità o di stati di esistenza diversi da quello che conosciamo. Anche la Scienza ha provato a dare delle risposte agli stessi interrogativi, ma la sua indagine si è limitata agli aspetti pratici del problema: ha prodotto delle interessanti teorie sulla genesi dell'Universo e sugli eventi che hanno portato alla nostra esistenza, ma non si è mai pronunciata su quello che potrebbe essere il nostro ruolo in tutto ciò, con le conseguenze di cui abbiamo parlato durante la lezione sulla memoria.
Il Maestro Canaro, che non riusciva ad accettare né i dogmi delle religioni tradizionali né lo scollamento fra uomo e Universo prodotto dalle ipotesi scientifiche, si pose una domanda:

> È possibile dare una spiegazione dell'Esistenza sfruttando solo ciò di cui abbiamo esperienza diretta?

La maggior parte delle religioni, per "funzionare", richiede da una a tre dimensioni aggiuntive, oltre quelle note; la Scienza, per le sue *super-stringhe* ha bisogno almeno di sette dimensioni aggiuntive, ovvero il doppio di quelle che servono per un Aldilà non spirituale. Esiste una spiegazione più semplice?
Non essendo né un filosofo né un mistico, approcciò lo sviluppo della sua dottrina come se fosse stata un sistema software. Per prima cosa fece un'analisi del "sistema in esercizio", evidenziandone i principali difetti; poi identificò delle vulnerabilità logiche delle religioni canoniche e definì delle linee-guida atte a prevenirle; infine, descrisse le caratteristiche del C'hi++, spiegando come queste avrebbero potuto risolvere alcuni dei problemi evidenziati in precedenza. Come scrisse nella Proposta, ci sono dei "bug" che possiamo considerare comuni a tutte le metafisiche:

- i dogmi, che sono le fondamenta delle dottrine, sono facilmente attaccabili perché non possono essere dimostrati, ma solo accettati per fede;
- una religione può avere delle difficoltà nel modificare la propria dottrina, anche quando è evidente che uno dei suoi dogmi è errato;
- la contestazione di un dogma causa quasi inevitabilmente una separazione e le separazioni è probabile che sfocino in conflitti.

ed altri, che possiamo considerare comuni agli esseri umani:

- la tendenza a difendere i propri principii anche con mezzi che contrastano con i principii stessi;
- la tendenza a influenzare la propria obiettività con le proprie speranze.

Per correggere o quanto meno mitigare questi problemi, la sua metafisica avrebbe dovuto:

- limitare il numero dei dogmi;
- limitare gli elementi metafisici e le accettazioni per fede;
- non proporsi come Unica Verità Incontestabile, ma come un'approssimazione sicuramente incompleta e perfettibile della Verità;
- riconoscere le contraddizioni della dottrina e analizzarle obiettivamente, anche se ciò porterà a modificare la dottrina stessa.

Il Maestro Canaro applicò allo sviluppo della sua *metafisica-non-metafisica* lo stesso approccio che adottava quando doveva realizzare un software. Ci sono due modi diversi di progettare un software: il primo consiste nell'analizzare tutti i sistemi che svolgono azioni simili, prendere il meglio di ciascuno e metterlo nel nuovo sistema; in alternativa, si può progettare il sistema da zero e solo quando se ne è definita per grandi linee la struttura, studiare le soluzioni adottate dagli altri, integrandole nel proprio programma se lo si ritiene utile. Il primo approccio è più rapido e sicuro, ma tende a produrre risultati ripetitivi; il secondo approccio è più complesso, sia in termini di analisi che di implementazione, ma facilita l'innovazione perché l'immaginazione dell'analista non è condizionata da ciò che ha visto.
Essendo un sostenitore del secondo metodo, il Maestro Canaro lo applicò anche al C'hi++ e, dopo alcuni di anni di studio, arrivò alla conclusione che non solo è possibile ipotizzare una cosmogonia quasi del tutto priva di elementi metafisici (non del tutto priva, perché, come vedremo in seguito, una dose minima di trascendenza è necessaria per garantire la buona funzionalità della dottrina), ma che i precetti di questa dottrina erano compatibili con molti principii delle religioni canoniche.

Il C'hi++ ereditò alcuni concetti proprii delle filosofie note al Maestro Canaro,

come il dualismo Gravità/Elettricità elaborato da Poe in *Eureka*, che lo aveva affascinato per il modo in cui trasformava una forza cieca e inspiegabile come la Gravità nell'intenzione, cosciente, di tutto ciò che esiste di tornare a essere Uno. D'altro canto, la dottrina del Maestro Canaro rinnegò alcuni concetti comuni a molte religioni, come la possibilità di sottrarsi al ciclo delle rinascite o la presenza di punizioni o premii *ad-personam*.
Così come quando si analizza il funzionamento di un software non ci si cura delle singole variabili, ma si pensa al flusso complessivo del sistema, così il C'hi++ vede l'esistenza non in termini di interazioni fra individui, ma come l'evoluzione del flusso dell'Energia dell'Uno all'interno della matrice tridimensionale degli spazioni. Per il C'hi++ non esistono né anime, né fiumi infernali e chi muore in mare non troverà ad accoglierlo Rán, nella sua birreria in fondo al mare, ma verrà semplicemente riciclato, come le aree di memoria RAM all'interno di un computer.
Le nostre esistenze sono incidentali; pensare di punirle o di premiarle non avrebbe senso e contrasterebbe con il principio generale che tutto ciò che esiste è la manifestazione di un'unica Entità. Come ti ho detto all'inizio di queste lezioni, non è possibile andare in Paradiso o all'Inferno da soli: qualunque cosa avvenga nell'Universo, ci riguarda tutti.
Questo però non vuol dire che il C'hi++ rifiuti tutti concetti delle religioni che lo hanno preceduto; anzi. Molti precetti del C'hi++ sono compatibili con precetti o idee appartenenti ad altre mistiche o filosofie e si tratta spesso di filosofie che il Maestro Canaro non conosceva, quando pose la basi della sua dottrina. Per esempio, il Maestro Canaro non lesse mai (con suo grande rammarico) la *Divina Commedia*; ciò non ostante, il C'hi++ ha un punto di contatto con la visione dantesca dell'Aldilà come conseguenza del pentimento. Dante mette in Purgatorio i peccatori che hanno capito di aver sbagliato, mentre condanna all'Inferno quelli che, malgrado tutto, non riescono a prendere coscienza delle proprie colpe. Come abbiamo detto in precedenza e come vedremo durante la lezione sul debug, il C'hi++ concorda con questa idea.
Similmente, ci sono diverse affinità fra i C'hi++ e la *Bhagavad-Gita*, anche se lui la lesse mentre stava redigendo la *Proposta*, quando i punti nodali del suo Credo erano già stati definiti.
Oltre alla citazione che ti ho fatto parlando del programmatore, ci sono dei brani che ricordano molto le affermazioni contenute in *Sostiene Aristotele*; per esempio, sulla natura dell'Universo:

> Alla fine del proprio ciclo d'esistenza, un mondo collassa su se stesso, riassorbendo in una massa tenebrosa ogni forma di manifestazione: esseri viventi e oggetti inanimati giacciono allo stato latente in una condizione caotica. I cicli cosmici sono periodi temporali chiamati Manvantara, suddivisi al proprio interno in quattro ere o Yuga, ciascuna caratterizzata da una particolare qualità dell'esistenza. Si tratta di un ritorno periodico a condizioni di vita non uguali ma analoghe, da un punto di vista qualitativo, a quelle dei cicli precedenti, una successione di quattro ere che ricorda, su scala ridotta,

l'alternarsi delle quattro stagioni.

O sul dualismo Gravità/Entropia :

> Il Sāṁkhya, la dottrina su cui si fonda lo Yoga, parla di due princìpi che, interagendo tra loro, manifestano l'intero universo con tutti gli esseri viventi e gli oggetti inanimati che lo popolano: Prakṛti, il polo materiale e femminile, e Puruṣa, quello spirituale e maschile; nell'essere umano Prakṛti costituisce il corpo e la mente, che diventano la dimora dell'anima individuale (puruṣa).

O su quelli che lui definiva: i *Post-It*:

> Ci sono due categorie di saṁskāra; la prima consiste nelle vāsanā, che sono impressioni lasciate nella mente dagli avvenimenti passati, tracce qui conservate allo stato latente ma pronte a manifestarsi in presenza delle condizioni adatte, cioè di situazioni analoghe a quelle che le hanno generate, e che le attiverebbero a causa della loro affinità. Sulla spinta delle vāsanā, una volta che siano attivate, e degli stati d'animo che queste manifestano, l'individuo presenta una tendenza inconscia ad agire in un determinato modo, e più in generale ad avere un certo tipo di comportamento, di sensibilità, di carattere; si tratta di una predisposizione innata che lo induce, nel bene come nel male, ad un comportamento analogo a quello che ha tenuto in passato, creando un circolo vizioso (o virtuoso) che si autoalimenta.

Puoi trovare delle analogie con i precetti del C'hi++ anche nel *Mantiq al-Tayr*:

> Tutto è un'unica sostanza in molteplici forme, tutto è un unico discorso in diverse espressioni (…) Egli sfugge a ogni spiegazione, a qualsiasi attributo. Di Lui soltanto una pallida idea ci è concessa, dare compiuta notizia di Lui è impossibile. Per quanto bene o male si parli di Lui, in realtà d'altri non si parla che di se stessi.

o anche:

> O Creatore, tutto il male o il bene che feci,
> in verità lo feci solo a me stesso.

Per certi versi anche la stessa Genesi biblica può essere considerata un'allegoria della cosmogonia spazionista: il Paradiso è l'Uno primigenio, mentre Adamo (*Puruṣa*) ed Eva (*Prakṛti*) sono l'Ente che ne causa la disgregazione, generando un Universo dove si partorisce nel dolore e dove ci si deve guadagnare il pane con il sudore della fronte.
Il Maestro Canaro pensava che tutto questo fosse normale. Come scrisse nel MANIFEST GitHub del C'hi++:

> Spogliate degli orpelli e ricondotte alle loro caratteristiche essenziali, le diverse ipotesi metafisiche hanno molti punti in comune perché so-

> no tutte, in una maniera o nell'altra, la risposta a uno stesso bisogno: la ricerca di una giustificazione alla nostra esistenza.

In una nota della mappa mentale su cui basò lo sviluppo iniziale della dottrina, aggiunse:

> Le diverse religioni, possono essere delle forme derivate di una stessa mistica iniziale? Esistono dei "dati membro" e delle funzioni comuni, che siano state ridefinite con il passare del tempo, ma che facciano capo a un corpo di credenze (o di nozioni) iniziale? Anche solo in questa mappa, se ne trovano diverse (p.es. Empedocle -> Poe). Così come le classi di un linguaggio Object-Oriented sono ridefinite per adattarsi a uno specifico contesto di utilizzo, così pure la Mistica iniziale potrebbe essere stata "overloaded" per adattarsi a uno specifico luogo o tempo. Se fosse così, tanto più si va indietro nel tempo, e quindi nella gerarchia di classi, tanto più ci si dovrebbe avvicinare alle caratteristiche proprie della Mistica. È possibile definire una gerarchia di classi figlie della classe astratta `Credo`? Semplificando molto (visto che sono le 3 di notte): Budda e Zoroastro influenzano i Greci, che influenzano gli Ebrei, che a loro volta influenzano i Cristiani, che alla fine producono i Testimoni di Geova… Allo stesso modo (sempre semplificando), dal C si è evoluto il C++ e dal C++, Java.

Solo alcuni anni dopo, annotò questa frase in un libro di Guenon:

> Il vero spirito tradizionale, quale si sia la forma da esso rivestita, è in fondo sempre e ovunque lo stesso; le forme diverse, specificamente adatte a queste o quelle condizioni mentali, a queste o quelle circostanze di tempo e di luogo, sono solo le espressioni di una unica e sola verità.

Fra il C'hi++ e le religioni canoniche c'è la stessa differenza che passa fra una mappa topografica e un'immagine da satellite.
Quel senza Dio di Dawkins, ha detto che:

> Uno dei caratteri di una folle stravaganza è un uso troppo entusiasta dell'analogia.

Una frase curiosa, da parte di un esponente di una setta che cerca di descrivere tutto ciò che esiste con analogie matematiche e nega l'esistenza di ciò che non riesce a convertire..
Entusiasmi a parte, le mappe e le immagini da satellite hanno diverse analogie con le discipline metafisiche. Anche le mappe e le immagini, come la metafisica, sono costrette a rappresentare il loro soggetto a un rapporto di scala ridotto e con due sole dimensioni in vece di tre (o di quattro se, oltre alla profondità, vuoi considerare anche il tempo). Anche le mappe e le immagini, per questo motivo, devono rappresentare il loro soggetto per mezzo di analogie: le carte

topografiche usano delle linee altimetriche e dei simboli; le immagini satellitari usano dei pixel o dei piccoli punti di colore. In nessuno dei due casi ciò che noi vediamo è davvero ciò che rappresenta; è il nostro cervello che decide di crederlo tale: nel caso della carta topografica, perché la legenda ci permette di definire una correlazione fra significato e significante; nel caso dell'immagine, perché il nostro occhio riconosce in quelle combinazioni di pixel o di punti di colore degli alberi, il mare o delle case.
Un'altra analogia, conseguenza dei due punti precedenti, è che è sbagliato confondere i simboli con ciò che rappresentano: i quadratini scuri delle mappe *non* sono case; i punti colorati delle immagini *non* sono un bosco. Mappe e immagini hanno senso solo a un certo livello di lettura; se lo oltrepassiamo, se cerchiamo di ottenere più informazioni o verosimiglianza avvicinando lo sguardo, otteniamo l'effetto opposto, perché i simboli si rivelano per quello che sono: punti colorati o linee su un foglio. Questo però non vuol dire che ciò che rappresentano sia falso, ma che noi non stiamo guardando con *il giusto paio di occhi*, come direbbe Hunter Thompson.
Il Maestro Canaro pensava che fosse per questo motivo che alcune religioni sono contrarie alla rappresentazione diretta della Divinità: perché è facile che poi si confonda il simbolo con ciò che rappresenta. Tornando al paragone iniziale, le religioni tradizionali sono delle immagini da satellite, mentre il C'hi++ è una mappa topografica.
Mentre i Credi religiosi riescono a riprodurre — nei limiti imposti dalla nostra condizione — tutta la bellezza del Creato, il C'hi++ si limita a darne una descrizione schematica, più povera di contenuti e di poesia, ma più facile da accettare per chi non abbia la benedizione della Fede. Un'immagine da satellite ha un valore contemplativo: è bella da guardare sullo schermo del tuo computer o anche da appendere al muro, come un quadro, ma se ti sei perso in un bosco o in mezzo ai monti, una mappa topografica, proprio in virtù della sua schematicità, ti permetterà più facilmente di ritrovare la strada di casa.
Il C'hi++ non cerca di rubare fedeli alle religioni canoniche. Non avrebbe senso: sarebbe come cercare di convincere chi sia già sposato con l'amore della sua vita a fare un matrimonio di interesse: se tu hai la Fede non hai bisogno di conferme razionali; possono compiacerti, ma non ti sono necessarie. Il C'hi++, però, può dare forza a quelle (tante) persone che *ancora credono in tutto ciò in cui più nessuno crede*, come li descrisse Longanesi; quella *Banda degli Onesti* che tutti i giorni fa il proprio dovere al meglio possibile anche se non gli conviene, anche tutto e tutti intorno a loro sembrano spingerli all'egoismo e all'indifferenza. Può aiutarli a non arrendersi e può insegnare loro che non è importante vincere le partite, ma giocare sempre meglio. Riconoscere gli sbagli che si sono fatti, imparare da essi e cercare di non ripeterli più, partita dopo partita, in una ricerca continua del meglio. Se si comporteranno così, qualunque sarà il loro lavoro, fosse anche pulire i cessi, sarà comunque Arte.

Gli stream

> Non puoi immergere i tuoi byte due volte nello stesso stream

Oggi ti parlerò degli *stream* che, com'è noto, sono la componente più importante del C++.

Il C++ eredita dal C l'assenza di parole chiave per la gestione dell'I/O. Al posto di istruzioni come la `print` del BASIC, utilizza delle librerie di classi e funzioni che permettono di convertire in testo stampabile gli oggetti gestiti dal programma o di convertire degli elementi testuali in oggetti. Non potrebbe essere altrimenti: il C++ non deve gestire solo stringhe e numeri, come il BASIC, ma anche numeri in virgola mobile, puntatori e soprattutto i tipi di dato definiti dall'utente, per i quali non sarebbe possibile definire un comportamento standard e che quindi dovrebbero essere trattati in maniera differente dai dati primitivi, con tanti saluti alla coerenza del linguaggio.
Oltre a poter sfruttare le librerie di funzioni del *C*, il C++ ha una propria libreria di I/O, basata sulla gerarchia delle classi `stream`, che permette di gestire anche i tipi di dato definiti dall'utente. Abbiamo visto degli esempii di questa caratteristica quando abbiamo parlato di polimorfismo e di overload degli operatori:

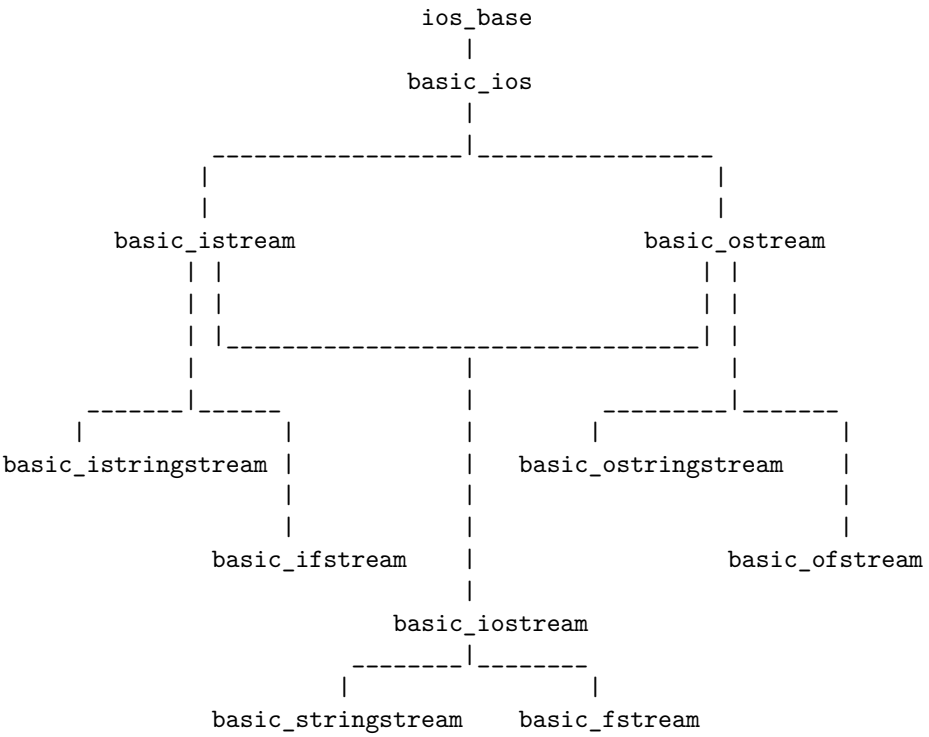
```
ostream& operator << (ostream& os, const Animale& animale) {
    os  << "Specie:" << animale.getSpecie() << "\t"
        << "Razza:"  << animale.getRazza()  << "\t"
        << "Sesso:"  << animale.getSesso()
        << endl;
    return os;
}
```

Questo codice "insegna" all'operatore `<<` come comportarsi per visualizzare un oggetto di classe `Animale`. Lo stesso si può fare (e lo abbiamo fatto) per qualsiasi altro tipo definito dall'utente. È la sintassi del linguaggio che si adatta alle esigenze del programmatore, e non viceversa.
Alcuni concetti chiave per la comprensione degli stream sono:

- uno *stream* è un'astrazione che rappresenta la sorgente o la destinazione di un insieme di dati di lunghezza variabile: l'input da tastiera, l'output su schermo, i buffer di memoria, le stringhe, i file;
- l'output su stream verso una qualsiasi destinazione, viene definito *scrittura* o *inserimento* e si effettua per mezzo dell'operatore `<<`;
- con i termini *lettura* o *estrazione*, invece, si intende l'operazione di acquisizione da una sorgente, effettuata dall'operatore `>>`.

La libreria `iostream` del C++ permette di gestire le operazioni di I/O su stream per mezzo di classi derivate da due classi base: `streambuf` e `iosbase`. La libreria ha due diverse "linee genealogiche": una destinata alla gestione dei caratteri di un byte e una destinata ai caratteri multi-byte. Le classi della libreria multibyte hanno lo stesso nome delle classi ordinarie, con l'aggiunta del prefisso: "w".

Questo è lo schema di ereditarietà delle classi della libreria `iostream`:



A parte `ios_base`, queste sono tutte classi template che sono poi istanziate con parametri differenti per gestire la gestione dei tipi di carattere `char` and `wchar_t`. Per esempio, la classe `ostream` è una specializzazione della classe `basic_ostream`:

```
typedef basic_ostream<char> ostream;
```

Il suo corrispettivo multi-byte è la classe `wostream`:

```
typedef basic_ostream<wchar_t> wostream;
```

La classe template `basic_ostream`, a sua volta, deriva da `basic_ios`:

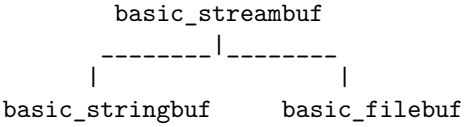
```
template<
    class CharT,
    class Traits = std::char_traits<CharT>
> class basic_ostream
: virtual public std::basic_ios<CharT, Traits>
```

che, a sua volta, deriva da `ios_base`:

```
template<
    class CharT,
```

```
    class Traits = std::char_traits<CharT>
> class basic_ios
: public std::ios_base
```

In sostanza: se davanti al nome c'è il prefisso `basic_`, si tratta della classe template; se c'è c'è la lettera "w", si tratta della versione multi-byte, altrimenti è la classe ordinaria.
Oltre alle classi derivate da `iosbase`, la libreria comprende anche delle classi per la gestione dei buffer di dati:



La classe template virtuale `basic_streambuf`, che fa parte della libreria, ma non della discendenza da `ios_base`, contiene i dati e le funzioni necessarie alla gestione di un buffer di caratteri. Le sue classi derivate `basic_stringbuf` e `basic_filebuf` sono invece specializzate, rispettivamente, nella gestione di buffer in memoria e su file. Anche in questo caso, la libreria comprende due versioni di ciascuna classe, specializzate per la gestione di `char` and `wchar_t`.

```
typedef streambuf  basic_streambuf<char>
typedef wstreambuf basic_streambuf<wchar_t>
typedef filebuf    basic_filebuf<char>
typedef wfilebuf   basic_filebuf<wchar_t>
```

Come forse avrai intuito, esaminare le singole classi della libreria `iostream` è un'attività che rivaleggia, in quanto a tedio, con l'epigrafia classica, ma ci permetterà di vedere applicati tutta una serie di principii di cui abbiamo parlato nelle lezioni precedenti, perciò, facciamoci forza e andiamo a incominciare.

La classe `ios_base` e la sua prima discendente `basic_ios` sono classi generiche che forniscono le funzioni di base per la gestione degli stream, indipendentemente dal fatto che si tratti di stream di input o di output.
Una peculiarità di `ios_base` è che non possiede un costruttore pubblico, quindi non è possibile utilizzarla per creare oggetti, ma solo come base per delle classi derivate.
Le istanze specializzate di `basic_ios` sono:

```
typedef basic_ios<char>    ios;
typedef basic_ios<wchar_t> wios;
```

Tramite i metodi di queste classi è possibile verificare o modificare lo stato interno dello stream, la sua formattazione o definire delle funzioni callback per la gestione dei dati.
Il dato membro `openmode`, per esempio, definisce il modo in cui debba essere aperto lo stream:

app	Fa sì che ogni operazione di output avvenga alla fine dello stream.
ate	In apertura dello stream, sposta il punto di inserimento al termine (***at e**nd*) del buffer di I/O.
binary	Gestisce il contenuto dello stream come un flusso di dati binario.
in	Permette operazioni di input.
out	Permette operazioni di output.
trunc	Azzera il contenuto dello stream all'apertura.

Il dato membro `iostate`, che utilizzeremo in uno dei prossimi esempii, contiene le informazioni sullo stato corrente dello stream:

goodbit	Nessun errore
eofbit	È stata raggiunta la fine dello stream.
failbit	L'ultima operazione di I/O è fallita.
badbit	L'ultima operazione di I/O non era valida.
hardfail	Si è verificato un errore irrecuperabile.

Entrambi questi dati membro sono delle bitmask, quindi possono contenere più di un valore. L'istruzione seguente, per esempio, apre uno stream su file combinando in `OR` tre possibili valori per `openmode`:

```
fstream file_io("io.txt"
               , ios_base::in | ios_base::out | ios_base::app);
```

Dopo `basic_ios`, le classi della libreria si specializzano nell'input o nell'output: da un lato `basic_istream`, da cui derivano i due stream standard di input `cin` e `wcin`; dall'altro `basic_ostream`, da cui derivano gli stream standard di output `cout`, `cerr`, `clog` e le loro controparti "wide": `wcout`, `wcerr`, `wclog`.
Da queste due classi generiche derivano delle classi template specializzate nell'input o nell'output su file o in memoria:

```
template
<class Elem, class Tr = char_traits<Elem>>
class basic_ifstream
: public basic_istream<Elem, Tr>

template
<class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>>
class basic_istringstream
: public basic_istream<Elem, Tr>

template
<class Elem, class Tr = char_traits<Elem>>
class basic_ofstream
: public basic_ostream<Elem, Tr>
```

```
template
<class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>>
class basic_ostringstream
: public basic_ostream<Elem, Tr>
```

e una classe capace di gestire entrambe le operazioni:

```
template
<class Elem, class Tr = char_traits<Elem>>
class basic_iostream
: public basic_istream<Elem, Tr>
, public basic_ostream<Elem, Tr>
```

anche questa, con due specializzazioni per la gestione di file e memoria:

```
template
<class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>>
class basic_stringstream
: public basic_iostream<Elem, Tr>

template
<class Elem, class Tr = char_traits<Elem>>
class basic_fstream
: public basic_iostream<Elem, Tr>
```

Prima che ci assalga un attacco di narcolessia, vorrei mettere in atto tutto questo con qualche esempio.
Abbiamo già visto diversi esempii di output su stream:

```
cout << "Hello World!"  // stringhe
     << 12              // interi
     << 0.35            // float
     << argv[1]         // puntatori
     << endl;
```

Non abbiamo ancora parlato dell'input da stream, che però ha un funzionamento piuttosto simile:

```
/**
 * @file src/stream-input.cpp
 * Programma di esempio per la gestione dell'input da stream.
 */

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
```

```
    string stringa;

    /** Scrive un messaggio sullo schermo */
    cout << "Inserire una stringa: ";

    /** Legge una stringa da tastiera */
    cin >> stringa;

    /** La scrive sullo schermo */
    cout << stringa << endl;

    return 0;
}
```

Questo codice legge una stringa dallo standard input e la scrive sullo schermo, ma ci mostra una peculiarità dell'input da stream:

```
> g++ src/cpp/stream-input-1.cpp -o src/out/esempio
> src/out/esempio
Inserire una stringa: Penso, quindi sono.
Penso,
```

Come vedi, le operazioni di lettura con l'operatore » si arrestano al primo carattere di spaziatura; perciò, se vogliamo leggere tutta la stringa, dobbiamo modificare il codice:

```
/**
 * @file src/stream-input-2.cpp
 * Programma di esempio per la gestione dell'input da stream.
 */

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    char   buffer[256];
    string stringa;

    /**
    *   Legge una stringa da tastiera tramite
    *   la funzione membro della classe istream.
    */
    cout << "Inserire una stringa: ";
    cin.getline(buffer, 256);
    cout << buffer << endl;
```

```
    /**
    *   Legge una stringa da tastiera tramite
    *   la funzione globale getline.
    */
    cout << "Inserire una stringa: ";
    getline(cin, stringa);
    cout << stringa << endl;

    return 0;
}
```

```
> g++ src/cpp/stream-input-2.cpp -o src/out/esempio
> src/out/esempio
Inserire una stringa: Penso, quindi sono.
Penso, quindi sono.
Inserire una stringa: Cogito ergo sum.
Cogito ergo sum.
```

Un'altra caratteristica degli operatori `<<` e `>>` è che la loro precedenza è minore di quasi tutti gli altri operatori, il che vi consente di scrivere delle istruzioni come questa:

```
cout << "Due più due fa: " << 2 + 2 << '\n' ;
```

Gli operatori logici di AND `|`, di OR inclusivo `&` e di XOR esclusivo `^`, hanno una precedenza minore degli operatori `<<` e `>>` e, se non vengono isolate tra parentesi, le operazioni che li coinvolgono possono essere causa di errori. Per esempio, in un'istruzione come la seguente, l'operatore `&` verrebbe interpretato come un riferimento a un oggetto, con conseguenze diverse da quelle attese:

```
cout << "Il valore è: " << 2 & 2 << '\n' ; // ERRORE!
```

la sintassi corretta è, invece:

```
cout << "Il valore è: " << (2 & 2) << '\n' ;   // OK
```

Il comportamento di default degli operatori di input da stream prevede anche delle convenzioni di formattazione:

- **il formato di conversione della base è decimale**;
- **il carattere di riempimento è lo spazio**;
- **la precisione delle cifre a virgola mobile è la stessa utilizzata da printf()**, con arrotondamento della sesta cifra decimale;
- **la larghezza del campo ha valore di default 0**, il che significa che lo stream di output utilizzerà tutti i caratteri necessari alla visualizzazione dell'intero valore o stringa.

Le prime tre modifiche sono permanenti: una volta impostati, i nuovi valori saranno validi fino a che un'altra istruzione non torni a modificarli; le modifiche alla larghezza del campo di input, invece, valgono solo per l'istruzione che le

richiede.
In alcuni esempii precedenti abbiamo visto che è possibile modificare il formato di output di default di uno stream tramite dei *maipolatori*:

```
cout << setfill('0') << setw(2) << ora._h << ":"
     << setfill('0') << setw(2) << ora._m << ":"
     << setfill('0') << setw(2) << ora._s << endl;
```

Lo stesso risultato si può ottenere per mezzo di apposite funzioni delle classi `ios_base` e `basic_ios`, che permettono di alterare il carattere di riempimento, la precisione delle cifre decimali e la larghezza del campo:

```
/**
 * @file src/stream-format.cpp
 * Formattazione dell'I/O con gli stream.
 */

#include <iostream>
#include <iomanip>

using namespace std;


int main(int argc, char** argv)
{
    double d = 123.456789 ;

    /** Mostra i valori di default */
    cout << endl;
    cout << "Valori di default" << endl;
    cout << "  width:     "  << cout.width()        << endl;
    cout << "  precision: "  << cout.precision()    << endl;
    cout << "  fill:      '" << cout.fill()  << "'" << endl;
    cout << "  output:    "  << d                   << endl;

    /** Modifica il formato e mostra il nuovo output */
    cout << endl;
    cout << "Modifica formato" << endl;
    cout << "  output:    "  ;
    cout.precision(4) ;
    cout.fill('#') ;
    cout.width(10) ;
    cout << d << endl;

    /** Mostra la persistenza dei valori */
    cout << endl;
    cout << "Valori correnti" << endl;
```

```
    cout << "  width:     "  << cout.width()        << endl;
    cout << "  precision: "  << cout.precision()    << endl;
    cout << "  fill:      '" << cout.fill()  << "'" << endl;
    cout << "  output:    "  << d                   << endl;

    return 0;
}
```

Se compili ed esegui questo codice, ottieni:

```
> src/out/esempio

Valori di default
  precision: 6
  fill:      ' '
  width:     0
  output:    123.457

Valori modificati
  output:    #####123.5

Valori correnti
  precision: 4
  fill:      '#'
  width:     0
  output:    123.5
```

Quando un'operazione di lettura o scrittura su stream fallisce, il valore del dato membro `iostate` assume un valore differente da zero. La classe basic_ios ha delle funzioni membro booleane che tornano `true` o `false` se il valore `iostate` indica un determinato evento e la funzione `rdstate` che torna il valore assoluto di `iostate`:

good	nessun errore: il valore di `iostate` è 0
eof	è stata raggiunta la fine del file
fail	c'è stato un errore di I/O non bloccante
bad	c'è stato un errore di I/O bloccante
rdstate	torna il valore corrente di `iostate`

Queste funzioni permettono di interrompere la lettura o la scrittura di uno stream quando si verifica un errore o se si è raggiunta la fine del file. Una cosa che non devi fare mai, però, è di utilizzare la funzione `eof` all'interno di un ciclo `while` per la lettura di un file:

```
/**
```

```
 * @file src/stream-eof.cpp
 * Gestione dell'I/O su file con gli stream.
 */

#include <iostream>
#include <fstream>

using namespace std;


int main(int argc, char** argv)
{
    ifstream testo;
    int      n;

    /** apre il file in lettura */
    testo.open(argv[1]);

    /** Verifica che il file sia aperto */
    if(testo.is_open()) {

        /** Utilizza eof per gestire il ciclo */
        while(!testo.eof()) {

            /** Legge un numero dal file */
            testo >> n;

            /** Lo scrive a video */
            cout << n << endl;
        }

    }

    /** Chiude il file di input */
    testo.close();

    return 0;
}
```

Se fai leggere a questo programma un file che contenga i numeri: 10, 20 e 30, otterrai questo output:

```
> g++ src/cpp/stream-eof.cpp -o src/out/esempio
> src/out/esempio src/cpp/stream-eof.txt
10
20
30
```

30

L'errore si verifica perché il controllo della funzione `eof` avviene prima della quarta operazione di lettura, quando lo stream è ancora in stato `good`. Un modo migliore di gestire questi casi è di utilizzare la funzione `good`, che ci permette di verificare anche la corretta apertura del file:

```
/**
 * @file src/stream-good.cpp
 * Gestione dell'I/O su file con gli stream.
 */

#include <iostream>
#include <fstream>

using namespace std;


int main(int argc, char** argv)
{
    ifstream testo;
    int      n;

    /** apre il file in lettura */
    testo.open(argv[1]);

    /**
    *   Il ciclo si ripete fino a che non
    *   si verifica un errore
    */
    while(testo.good()) {

        /** Legge un numero dal file */
        testo >> n;

        /** Si interrompe se il file è finito */
        if(testo.eof()) break;

        /** Altrimenti, scrive il numero */
        cout << n << endl;
    }

    /** Chiude il file di input */
    testo.close();

    return 0;
}
```

Se compili ed esegui questo programma, ottieni il risultato corretto:

```
> g++ src/cpp/stream-good.cpp -o src/out/esempio
> src/out/esempio src/cpp/stream-eof.txt
10
20
30
```

Le *eccezioni* permettono di gestire gli errori che avvengono durante l'esecuzione del programma. Quando succede qualcosa di anormale, il sistema *lancia* un'eccezione, ovvero trasferisce il controllo del processo dalla funzione corrente a blocchi di istruzioni specifici, chiamati *exception handler*. Perché tutto questo avvenga, il codice che genera l'errore deve essere racchiuso in un blocco `try`/`catch`:

```
try {

    // codice che potrebbe dare errore

} catch (...) {

    //  istruzioni per la gestione dell'errore
}
```

Le eccezioni possono essere lanciate e gestite sia da codice specifico all'interno del programma, sia dai meccanismi automatici del C++:

```
/**
 * @file src/stream-eccezioni-1.cpp
 * Programma di esempio per la gestione delle eccezioni.
 */

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char** argv)
{
    ifstream testo;

    /**
    *   Fa sì che, se sia generata un'eccezione
    *   in caso di errore nella gestione del file.
    */
    testo.exceptions ( ios_base::failbit );
```

```
    /** Questa istruzione genererà un'eccezione */
    testo.open("fileinesistente.txt");

    testo.close();

    return 0;
}
```

Se compili ed esegui questo codice, causerai un errore che, non essendo gestito dal programma, è gestito dalla funzione standard del C++:

```
> g++ src/cpp/stream-eccezioni-1.cpp -o src/out/esempio
> src/out/esempio
libc++abi: terminating with uncaught exception of type
std::__1::ios_base::failure: ios_base::clear
: unspecified iostream_category error
zsh: abort      src/out/esempio
```

Se però inseriamo il codice che apre il file in un blocco `try`/`catch` e definiamo un *handler* per la gestione degli errori in apertura dei file, il risultato sarà più controllato:

```
/**
 * @file src/stream-eccezioni-2.cpp
 * Programma di esempio per la gestione delle eccezioni.
 */

#include <iostream>
#include <fstream>

using namespace std;


int main(int argc, char** argv)
{
    ifstream testo;

    try {

        testo.exceptions ( ios_base::failbit );
        testo.open("fileinesistente.txt");
        testo.close();

    } catch(ios_base::failure e) {

        cerr << "Errore in esecuzione" << endl;

    }
```

```
    return 0;
}
```

```
> g++ src/cpp/stream-eccezioni-2.cpp -o src/out/esempio
> src/out/esempio
Errore in esecuzione
```

Possiamo addirittura prevenire gli errori in apertura del file facendo sì che sia lo stesso programma a lanciare un'eccezione se si accorge che manca il nome del file nei parametri di avvio:

```
/**
 * @file src/stream-eccezioni-3.cpp
 * Programma di esempio per la gestione delle eccezioni.
 */

#include <iostream>
#include <fstream>

using namespace std;


int main(int argc, char** argv)
{
    ifstream testo;

    try {

        /** Verifica che esista il nome del file da aprire */
        if(argc < 2)
            throw "Nome file mancante";

        testo.exceptions ( ios_base::failbit );
        testo.open(argv[1]);
        testo.close();

    } catch(ios_base::failure e) {

        cerr << "Errore in esecuzione" << endl;

    } catch(char const* msg) {

        cerr << msg << endl;
    }

    return 0;
```

```
}
```

In questo modo, il programma è in condizione di gestire tutti i possibili errori di esecuzione:

```
> g++ src/cpp/stream-eccezioni-3.cpp -o src/out/esempio
> src/out/esempio
Nome file mancante
> src/out/esempio nomefile.txt
Errore in esecuzione
```

La libreria standard del C++ ha una classe specifica per la gestione delle eccezioni:

```
class exception {
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
}
```

Definendo una classe derivata da `exception` con altri dati membro e una funzione `what` specializzate, è possibile gestire in maniera più strutturata le segnalazioni di errore. È quello che faremo nel prossimo esempio.

```
/**
 * @file src/stream-eccezioni-4.cpp
 * Programma di esempio per la gestione delle eccezioni.
 * Richiede, in input, il numero di caratteri da leggere
 * e il path del file di input:
 *
 *    src/out/esempio <n caratteri da leggere> <file di input>
 */

#include <iostream>
#include <fstream>
#include <exception>

using namespace std;

/** Codici e stringhe di errore */
#define ERR_NONE          0
#define ERR_PARAMETRI   -10
#define ERR_FILE_OPEN   -20
#define S_SINTASSI      "USO: esempio <n caratteri> <path file>"
#define S_ERR_PARAMETRI "Errore nei parametri di input."
#define S_ERR_FILE_OPEN "Impossibile aprire il file di input"
```

```
/**
*   Definisce una classe derivata da exception
*   per la gestione degli errori.
*/
class Eccezione: public exception
{
private:
    int         _codice;
    const char* _errore;
public:

    /** Costruttore */
    Eccezione(int codice, const char* errore)
    : _codice(codice), _errore(errore) {
    }

    /** Funzione virtuale pura: va ridefinita */
    virtual const char* what() const throw() {
        return _errore;
    }

    /** Funzioni di interfaccia */
    int getCodice() { return _codice; }
    const char* getErrore() { return _errore; }

    /** Ridefinizione dell'operatore di output */
    friend ostream& operator<< (ostream& os, Eccezione e){
        os << e._codice << ": " << e._errore << endl;
        return os;
    }
};

int main(int argc, char** argv)
{
    ifstream testo;
    testo.exceptions ( ios_base::badbit );

    try {

        char c     = 0;
        int  letti = 0;

        /**
        *   Verifica che ci siano sia il nome del file di input
        *   che il numero di caratteri da leggere.
```

```
*/
if (argc < 3)
    throw Eccezione(ERR_PARAMETRI, S_ERR_PARAMETRI);

/** Definisce il numero di caratteri da leggere */
int da_leggere = atoi(argv[1]);

/**
*   Imposta la exception mask dello stream per fare
*   sì che un errore di I/O generi un'eccezione,
*   poi apre il file in lettura.
*   Usa un blocco try/catch per intercettare una
*   eventuale eccezione e gestirla in maniera
*   omogenea al resto del codice.
*/
try {
    testo.exceptions ( ios_base::badbit
                     | ios_base::failbit );
    testo.open(argv[2]);
} catch(ifstream::failure e) {
    throw Eccezione(ERR_FILE_OPEN, S_ERR_FILE_OPEN);
}

/**
*   Re-imposta la exception mask per evitare
*   eccezioni a fine file.
*/
testo.exceptions ( ios_base::goodbit);

/**
*   Legge il testo e lo stampa a video
*   Se è stato definito un numero massimo di
*   caratteri, si ferma lì.
*/
while(testo.good()) {
    if((c = testo.get()) != EOF) {
        letti++;
        cout << c;
    }
    if((da_leggere != 0) && (letti >= da_leggere)) {
        cout << endl;
        break;
    }
}

/** Chiude il file di input */
```

```
        testo.close();

    } catch (Eccezione e) {

        /** Stampa a video l'eccezione */
        cerr << e << endl;

        /** Mostra la sintassi di chiamata **/
        cerr << S_SINTASSI << endl;

        /** Esce con un codice di errore */
        exit(e.getCodice());
    }

    return 0;
}
```

Se compili ed esegui questo codice, otterrai il seguente output, a seconda dei parametri forniti:

```
> g++ src/cpp/stream-eccezioni-4.cpp -o src/out/esempio
> src/out/esempio
-10: Errore nei parametri di input.
USO: esempio <n caratteri> <path file>

> src/out/esempio 41
-10: Errore nei parametri di input.
USO: esempio <n caratteri> <path file>

> src/out/esempio 41 src/cpp/stream-input.txt
Nacqui da famiglia ricca, ma troppo tardi

> src/out/esempio 0 src/cpp/stream-input.txt
Nacqui da famiglia ricca, ma troppo tardi.
Secondogenito, vidi la florida impresa paterna andare in dote
per diritto di nascita, ma anche per naturale inclinazione
ai miei monozigotici fratelli maggiori e, com'è consuetudine
per i figli cadetti, fui avviato alla vita monastica.
Entrai in seminario all'età di nove anni e presi i voti il
giorno del mio diciottesimo compleanno.
Conobbi il Maestro quattro anni dopo.
```

Il Maestro Canaro diceva che una filosofia, per spingere i suoi seguaci a comportarsi in maniera corretta, deve possedere due caratteristiche: *trascendenza* e *permanenza*.
Se, per un caso o per volere del Cielo, gli esempii che ti ho fatto finora pren-

dessero coscienza di sé, senza però sapere di far parte di una serie di lezioni, probabilmente si sentirebbero inutili e sciocchi. Perfino l'ultimo esempio che abbiamo visto, che è il più complesso di tutti, non potrebbe fare a meno di chiedersi quale sia il senso della sua esistenza, dato che lo stesso risultato si può ottenere con una semplice istruzione da riga di comando:

```
head -c 41 src/cpp/stream-input.txt
```

Qualche esempio riuscirebbe comunque a fare il proprio dovere, ma ce ne sarebbero altri che reagirebbero male a questa epifania: i più deboli si deprimerebbero, mentre i più ambiziosi cercherebbero una compensazione nell'accumulo eccessivo di risorse di sistema: RAM, spazio disco o cicli CPU.
Al contrario, se gli esempii sapessero di essere parte integrante di una serie di lezioni, tutto ciò che altrimenti appare insensato o inutile, dai commenti pleonastici fino al parametro numerico dell'ultimo esempio, acquisterebbe il giusto significato e ciascun esempio saprebbe di essere non solo utile, ma necessario.
Se c'è una cosa che sappiamo per certa dell'Universo in cui viviamo è che si sta espandendo. Se nulla interverrà a mutare questo stato di cose, tutto ciò che esiste, dagli esseri viventi alle stelle, è destinato o prima o poi a spegnersi nella vittoria di Pirro dell'Entropia come un computer portatile a cui si scarichi la batteria.
Al contrario, se la Gravità riuscirà a invertire il moto delle galassie, tutto ciò che esiste, dalle stelle agli esseri viventi, è destinato o prima o poi ad annichilirsi nell'Uno in attesa di un nuovo ciclo di esistenza.
Questo, però, non basterà da solo a dare un senso alle nostre esistenze, perché, senza persistenza, le nostre azioni saranno come degli oggetti di classe `streambuf`: una volta spento il computer, non esisteranno più e tutto ciò che abbiamo fatto, giusto o sbagliato che sia, non avrà alcuna influenza su ciò che accadrà successivamente.
L'unica cosa che può salvarci dall'oblio e dall'insensatezza sono i `Post-It`, la persistenza.
Per dare un senso alla nostra esistenza abbiamo bisogno di un *hard-disk* su cui salvare gli stream delle nostre vite, in modo che ogni ciclo di esistenza possa fare tesoro delle esperienze passate. Senza di esso, Hitler varrà quanto Ghandi e Albert Schweitzer quanto Ted Bundy. Dovrà essere però un *hard-disk* meta-fisico, per sfuggire al *Big Crunch*, e questo ci riporta all'importanza della trascendenza.
Se mi guardo indietro, per il mezzo secolo su cui ho visibilità diretta, vedo una lunga serie di fallimenti ideologici. Il Sessantotto ha spazzato via delle parti sicuramente rivedibili, ma fondamentali della nostra Società senza darci nulla in cambio, tranne la minigonna. La lotta armata degli anni settanta ha sparato alle persone sbagliate, mentre la *reaganomics* è crollata alla fine degli anni '80 insieme al muro di Berlino. Le speranze degli anni '90 si sono schiantate l'11 Settembre 2001 sulle Torri Gemelle e anche Internet, che nelle intenzioni iniziali sarebbe dovuta essere un mezzo per dare a tutti la possibilità di esprimere le proprie idee si è trasformata, nel tempo, in un sistema di controllo e di disinformazione di massa.

Quel poco che restava dei nostri valori e delle nostre idee è stato annichilito dagli *smart-phone* e dai *social-network*.
Esiste un fattore comune alle ideologie degli ultimi cinquant'anni che ne ha accelerato l'obsolescenza e le ha rese incapaci di sopravvivere alla prima sconfitta: il rifiuto più o meno accanito di ogni forma non strumentale di spiritualità.
Se si definisce uno schema di valori negando allo stesso tempo qualsiasi forma di trascendenza, si è costretti a ricercare i valori e le motivazioni della propria etica all'interno dello schema stesso. Si può fare, ma è sbagliato e limitativo. È sbagliato, perché le regole che si definiscono sono sempre una conseguenza di esigenze contingenti (guerre, sopraffazioni, disparità sociali), venendo a mancare le quali lo schema logico del sistema perde di significato e si disgrega. È limitativo, perché restringe il numero dei possibili obiettivi da perseguire a un insieme finito di azioni o traguardi, raggiunti i quali non esiste più possibilità di migliorare.
Pensa al gioco degli Scacchi: non esiste nessun motivo, all'interno della scacchiera, che costringa ciascun pezzo a muoversi solo in una specifica maniera. Le torri si muovono in orizzontale, gli alfieri in diagonale e il cavallo salta con una traiettoria a "L" in ossequio a delle regole definite al di fuori della scacchiera, ma è proprio da queste limitazioni che deriva il fascino del gioco. Al contrario, la Società moderna è una scacchiera in cui ciascun pezzo si muove nella maniera che preferisce perché, in ossequio a un malinteso senso di libertà, sono state eliminate tutte le regole. Le persone di successo che si privano della vita sono pedoni che, arrivati alla fine della scacchiera grazie alla loro abilità, hanno scoperto che non esiste alcuna forma di promozione, perché insieme alle regole sono stati aboliti anche i giocatori.

Il Maestro Canaro una volta mi disse:

> Io non credo alla storia della conversione. David Chapman ha ucciso John Lennon perché pensava che sopra di lui ci fosse "only sky" e che solo così, avrebbe potuto dare un senso alla sua esistenza.

Il debug

> Cento Mondi di peccato sono dissipati dalla luce di un solo ticket

Il debug potrà non essere il senso della vita, come recita il titolo del tuo libro, ma è indubbiamente l'aspetto più importante della programmazione.
Nella tua carriera di programmatore potrai non utilizzare mai una classe *functor* o ridefinire l'operatore `->`, ma sicuramente farai degli errori e li dovrai correggere. Malgrado ciò, i manuali di programmazione non parlano mai del *debug*. Si sono scritte migliaia di pagine sui diversi linguaggi di programmazione; non

c'è primavera che non veda fiorire un nuovo paradigma di programmazione — strutturata, *object-oriented*, *agile*, *fuzzy* — eppure, nessuno si è mai preoccupato di formalizzare il processo di correzione del codice.
Non a caso, la decadenza del software è iniziata quando le stampanti laser hanno soppiantato le vecchie stampanti ad aghi. Il codice non si può leggere su un foglio A4: a meno che non sia un programma banale, non c'entrerà né in altezza né in larghezza. Il modulo in continuo di una stampante ad aghi a 136 colonne, al contrario, ti permette di stampare tutto il tuo codice e di rileggerlo con calma; correggerlo, se necessario e migliorarlo se possibile. È così che si facevano le revisioni di codice, quando c'erano il tempo e i soldi per fare le revisioni di codice.

Trascurare il debug è come affermare che non si faranno mai errori. Non importa quanto sia efficiente il linguaggio di programmazione; non importa quanto sia astuto e vigile il compilatore; non importa nemmeno quanto sia bravo il programmatore: o prima o poi, la distrazione, la stanchezza o un evento esterno permetteranno a un errore di intrufolarsi nel codice. Un puntatore utilizzato impropriamente, un ciclo in più o in meno in un'istruzione `for` o una virgola dimenticata fra i parametri di una `printf`: qualunque cosa sia, se il compilatore non sarà in grado di riconoscerla, finirà nel programma è resterà lì in attesa di produrre i suoi effetti dannosi. Il programma potrà funzionare correttamente per anni, ma poi, un bel giorno, qualcosa non andrà come sarebbe dovuto andare e a quel punto bisognerà analizzare il codice per trovare l'errore.
Fare degli errori è inevitabile, ed è importante sapere come porvi rimedio. Ancora più importante, però, è sapersi accorgere degli errori. Così come *l'Ikebarba inizia nel negozio*, il debug comincia nel momento in cui si scrive il codice. Il modo migliore per evitare che il codice contenga degli errori è scrivere del buon codice. Scrivere del buon codice vuol dire fare sempre il meglio che ti è possibile. Non salvare mai un file se non sei certo che funzionerà come deve e cerca sempre di pensare a cos'altro potrebbe fare il tuo codice, oltre a quello che vuoi tu. Come di ho detto in una delle nostre prime chiacchierate, il Buon Programmatore non si accontenta della strada più rapida, ma cerca sempre quella più efficiente e sicura, perché sa che scrivere del buon codice costa meno che riparare del codice fatto male. Il Maestro Canaro, una volta disse:

> Il Buon Programmatore è come un marinajo che, prima di un lungo viaggio, verifica tutta la sua attrezzatura di coperta, smontando ogni singolo elemento e sostituendo tutto ciò che potrebbe rompersi, perché non sa cosa succederà una volta che sarà in mare.

Il Buon Programmatore non può farne a meno, perché è nella sua natura; è questa, la differenza fra chi *fa* il programmatore e chi *è* un programmatore.

Gli errori del software possono essere di tre tipi:

- gli errori che si manifestano durante la fase di compilazione;
- gli errori che si manifestano durante l'esecuzione del programma;
- gli errori di analisi.

Gli **errori di compilazione** sono causati da costrutti incorretti, che bloccano il processo di creazione del programma. Un errore che farai spesso è di dimenticare il punto e virgola alla fine della dichiarazione di una classe:

```
class C {
private:
    float _raggio;
    float _area;
public:
    C (int m) : _raggio(m) {}
    float getRaggio() { return _raggio; }
    float area() {
       Quadrato q;
       return 3.14159 * q(_raggio);
    }
}
```

una banale distrazione che causerà immancabilmente il messaggio:

```
> g++ src/cpp/debug-errori.cpp  -c -o src/out/esempio
src/cpp/debug-errori.cpp:28:2: error: expected ';' after class
}
 ^
 ;
1 error generated.
```

Gli errori di compilazione sono i più facili da gestire, perché è il compilatore stesso a dirti quale sia il problema e in quale punto del codice si trovi. L'unica difficoltà che potresti avere, specie se stai lavorando con la `STL`, è decifrare i messaggi del compilatore:

```
In file included from /Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/iostrean
In file included from /Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/ios:216:
In file included from /Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/__locale
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/string:2027:19: error: no viab
      'const std::__1::basic_string<char>' to 'std::__1::basic_string<char, std::__1::char_t
      std::__1::allocator<char> >::value_type' (aka 'char')
        push_back(*__first);
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/string:2075:5: note: in instan
      template specialization 'std::__1::basic_string<char, std::__1::char_traits<char>, std
      >::__init<std::__1::istream_iterator<std::__1::basic_string<char>, char, std::__1::cha
      requested here
    __init(__first, __last);
    ^
src/cpp/debug-errori.cpp:43:12: note: in instantiation of function template specialization
```

```
      'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
      >::basic_string<std::__1::istream_iterator<std::__1::basic_string<char>, char, std::_
      long> >' requested here
    string s(begin,end);
           ^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/string:876:5: note: candidate
    operator __self_view() const _NOEXCEPT { return __self_view(data(), size()); }
    ^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/string:1055:31: note: passing
      '__c' here
    void push_back(value_type __c);
```

Se un costrutto è formalmente corretto, ma *potrebbe* essere un errore, il compilatore può segnalarlo con un **warning**, un messaggio di avviso che non blocca la compilazione, ma richiama l'attenzione del programmatore sull'anomalia.
Ho usato il condizionale (*può segnalarlo*) perché la notifica dei *warning* è un'opzione che deve essere attivata dal programmatore, specificando, fra i parametri di compilazione, quali avvisi vuole ricevere. Dato che i parametri per attivare i diversi tipi di *warning* sono davvero tanti:

```
-Waddress
-Wbool-compare
-Wbool-operation
-Wchar-subscripts
-Wcomment
-Wformat
-Wformat-overflow
-Wformat-truncation
-Wint-in-bool-context
-Winit-self
-Wlogical-not-parentheses
-Wmaybe-uninitialized
-Wmemset-elt-size
-Wmemset-transposed-args
-Wmisleading-indentation
-Wmismatched-dealloc
-Wmismatched-new-delete
-Wmissing-attributes
-Wmultistatement-macros
-Wnarrowing
-Wnonnull
-Wnonnull-compare
-Wopenmp-simd
-Wparentheses
-Wpessimizing-move
-Wpointer-sign
-Wrange-loop-construct
```

```
-Wreorder
-Wrestrict
-Wreturn-type
-Wsequence-point
-Wsign-compare
-Wsizeof-array-div
-Wsizeof-pointer-div
-Wsizeof-pointer-memaccess
-Wstrict-aliasing
-Wstrict-overflow=1
-Wswitch
-Wtautological-compare
-Wtrigraphs
-Wuninitialized
-Wunknown-pragmas
-Wunused-function
-Wunused-label
-Wunused-value
-Wunused-variable
-Wvolatile-register-var
-Wzero-length-bounds
```

e che possono essere tutti rilevanti nel processo di creazione del codice, la cosa migliore che puoi fare è di attivarli globalmente, con il parametro: `-Wall`

```
/**
 * @file src/debug-errori-warning.cpp
 * Tipi di errore e di warning.
 */

#include <iostream>

using namespace std;

class Quadrato
{
public:
    double operator()(double x) { return x * x; }
};

class C {
private:
    float _raggio;
    float _area;
public:
    C (int m) : _raggio(m) {}
    float getRaggio() { return _raggio; }
```

```
    float area() {
       Quadrato q;
       return 3.14159 * q(_raggio);
    }
};

int main(int argc, char** argv)
{
    C c(10);
    cout << c.area() << endl;
    return 0;
}
```

Se compili questo codice senza attivare i *warning*, il compilatore non ti segnala nulla di anomalo:

```
> g++ src/cpp/debug-errori-warning.cpp -c -o src/out/esempio
>
```

Se però aggiungi il parametro `-Wall` alla riga di comando, scopri che il parametro `_area` della classe `C` non viene utilizzato:

```
> g++ src/cpp/debug-errori-warning.cpp -Wall -c -o src/out/esempio
src/cpp/debug-errori-warning.cpp:19:11:
    warning: private field '_area' is not used
        [-Wunused-private-field]
    float _area;
          ^
1 warning generated.
```

Il codice è stato compilato, perché questa potrebbe essere una scelta intenzionale, ma il sistema ti notifica comunque l'anomalia, in modo che tu possa decidere se mantenerla o eliminarla.
Aggiungendo il parametro `-Wextra`, ottieni un *warning* aggiuntivo perché i parametri della funzione `main` non sono utilizzati:

```
> g++ src/cpp/debug-errori-warning.cpp -Wall -Wextra -o src/out/esempio
src/cpp/debug-errori-warning.cpp:29:14:
    warning: unused parameter 'argc'
        [-Wunused-parameter]
int main(int argc, char** argv)
             ^
src/cpp/debug-errori-warning.cpp:29:27:
    warning: unused parameter 'argv'
        [-Wunused-parameter]
int main(int argc, char** argv)
                          ^
src/cpp/debug-errori-warning.cpp:19:11:
    warning: private field '_area' is not used
```

```
        [-Wunused-private-field]
    float _area;
          ^
3 warnings generated.
```

Oltre a quelli definiti dai parametri `-Wall` e `-Wextra`, il compilatore prevede una lunga lista di *warning* che possono essere definiti individualmente, a seconda delle esigenze del programma. Per esempio, il parametro `-Wdouble-promotion` segnala quando una variabile di tipo `float` è promossa implicitamente a `double`:

```
> g++ src/cpp/debug-errori-warning.cpp -Wall -Wextra \
    -Wdouble-promotion -c -o src/out/esempio
src/cpp/debug-errori-warning.cpp:25:27:
    warning: implicit conversion increases
        floating-point precision: 'float' to
      'double' [-Wdouble-promotion]
       return 3.14159 * q(_raggio);
                        ~ ^~~~~~
src/cpp/debug-errori-warning.cpp:29:14:
    warning: unused parameter 'argc' [-Wunused-parameter]
int main(int argc, char** argv)
             ^
src/cpp/debug-errori-warning.cpp:29:27:
    warning: unused parameter 'argv' [-Wunused-parameter]
int main(int argc, char** argv)
                           ^
src/cpp/debug-errori-warning.cpp:19:11:
    warning: private field '_area' is not used
        [-Wunused-private-field]
    float _area;
          ^
4 warnings generated.
```

Molti programmatori ignorano i *warning*, pensano che se il programma può essere compilato non ci sia nient'altro di cui preoccuparsi. Tu non fare questo errore: nessun avviso deve essere ignorato.

Solo un programma formalmente ineccepibile può dare luogo a **errori di esecuzione**. Se non fosse formalmente ineccepibile, infatti, non sarebbe stato compilato e non potrebbe essere eseguito.
Gli errori di esecuzione sono tanto più pericolosi quanto più i loro effetti sono lievi. Un errore che causi il blocco del sistema sarà certamente rilevato e corretto; al contrario, un leggero errore di calcolo potrebbe passare inosservato e quindi causare grandi problemi.
Se dimentichi l'operatore di incremento all'interno di un ciclo `while` produrrai un ciclo infinito, che certamente attirerà la tua attenzione:

```
/**
 * @file pianeti-while-errore.cpp
 * Gestione dei Pianeti con errore evidente.
 */

#include <iostream>
#include "pianeti.h"

using namespace std;

int main(int argc, char** argv)
{
    int    p  = POS_MERCURIO;
    string nome;

    while (!(nome = nomePianeta(p)).empty()) {
        cout << nome << endl;
    }

    return 0;
}
```

```
> g++ src/cpp/pianeti-while-errore-1.cpp \
      src/cpp/pianeti-2.0.cpp \
   -o src/out/esempio
> src/out/esempio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
Mercurio
...
```

Se invece incrementi la variabile nel modo sbagliato:

```
/**
 * @file pianeti-while-errore.cpp
 * Gestione dei Pianeti con errore nel ciclo while
 */

#include <iostream>
```

```
#include "pianeti.h"

using namespace std;

int main(int argc, char** argv)
{
    int    p  = POS_MERCURIO;
    string nome;

    while (!(nome = nomePianeta(++p)).empty()) {
        cout << nome << endl;
    }

    return 0;
}
```

causerai un errore che, in questo caso, è evidente, perché all'elenco manca Mercurio, ma che in un programma più complesso potrebbe essere difficile da individuare:

```
> g++ src/cpp/pianeti-while-errore-2.cpp \
      src/cpp/pianeti-2.0.cpp \
   -o src/out/esempio
> src/out/esempio
Venere
Terra
Marte
Giove
Saturno
Urano
Nettuno
Plutone
```

Gli errori di esecuzione possono essere di due tipi: quelli che si manifestano in maniera deterministica e quelli che si manifestano in maniera casuale. L'errore nell'elenco dei pianeti si manifesterà a ogni esecuzione del programma e sarà quindi (relativamente) facile da individuare. Al contrario, questo codice produrrà un errore solo in determinate condizioni:

```
/**
 * @file debug-errore-stocastico.cpp
 * Errore che si verifica solo in deterinate circostanze.
 */

#include <iostream>
#include <fstream>

using namespace std;
```

```
#define ERR_FILE_NONE       -10
#define ERR_FILE_OPEN       -20
#define S_ERR_FILE_NONE     "Definire un file di input"
#define S_ERR_FILE_OPEN     "Impossibile aprire il file di input"
#define N_CHAR_MIN          300
#define N_BANNER_MAX        3
#define PUNTO               '.'

/**
*   Aggiunge un banner dopo ogni punto,
*   ogni N_CHAR_MIN caratteri.
*/
int banner_testo(ifstream& testo)
{
    int  n_banner = 1;
    int  letti = 0;
    char c = 0;

    /**
    *  Legge tutto il contenuto del file di input
    */
    while(testo.good()) {

        /**
        *   Legge il cotenuto del file,
        *   carattere per carattere
        */
        if((c = testo.get()) != EOF) {

            /** Incrementa il numero di caratteri letti */
            letti++;

            /** Scrive il carattere letto */
            cout << c;

            /**
            *   Se il carattere corrente è un punto
            *   e ha letto almeno N_CHAR_MIN caratteri
            *   e ha ancora banner da aggiungere,
            *   inserisce il codice del banner nel testo.
            */
            if(c == PUNTO
            && letti >= N_CHAR_MIN
            && n_banner <= N_BANNER_MAX) {
                cout << endl
```

```
                    << "<div id=\"banner-"
                    << n_banner
                    << "\"></div>"
                    << endl;
                n_banner++;
                letti = 0;
            }
        }
    }

    cout << endl;

    return n_banner;
}

int main(int argc, char** argv)
{
    ifstream testo;

    /** Verifica che ci sia il nome del file di input */
    if(argc < 2) {
        cerr << S_ERR_FILE_NONE << endl;
        exit(ERR_FILE_NONE);
    }

    /** Apre il file in lettura */
    testo.open(argv[1]);
    if(!testo.good()) {
        cerr << S_ERR_FILE_OPEN << endl;
        exit(ERR_FILE_OPEN);
    }

    /** Elabora il testo */
    banner_testo(testo);

    /** Chiude il file */
    testo.close();

    return 0;
}
```

La funzione `banner_testo` inserisce il *tag* HTML di un banner all'interno del testo di una pagina Web. I banner devono essere posizionati dopo un punto fermo, a distanza di almeno `N_CHAR_MIN` caratteri l'uno dall'altro.
Questo codice funziona correttamente con alcuni tipi di testo:

> Essere un ossessivo-compulsivo con una leggera tendenza alla para-

> noia, se ti guadagni da vivere facendo l'esperto di sicurezza, è un bene; le medesime peculiarità caratteriali, al contrario, sono decisamente un male quando alle 21:55 la tua donna di servizio ti scrive: "Ho fatto un molecolare e sono risultata positiva.
>
> Non potrai farti un tampone prima delle 8:00 dell'indomani quindi sai che ti aspettano almeno dieci ore di panico controllato; qualcuna di meno, se riesci ad addormentarti. Cerchi di distrarti guardando la televisione, ma l'ennesimo thriller con Jason Statham, intervallato da pubblicità di ansiolitici (un conflitto di interessi che ti riprometti di studiare con più attenzione, se sopravvivi), non fa che aumentare la tua agitazione; così, spegni il televisore, ti prepari una tisana relax, leggi un po' e poi cerchi di dormire.

Il codice della funzione `banner_testo`, però, è troppo ottimistico e delle piccole variazioni nel file di input, come l'aggiunta di punti di sospensione o di una URL, *potrebbero* causare degli errori nel posizionamento dei banner:

> Essere un ossessivo-compulsivo con una leggera tendenza alla paranoia, se ti guadagni da vivere facendo l'esperto di sicurezza, è un bene; le medesime peculiarità caratteriali, al contrario, sono decisamente un male quando alle 21:55 la tua donna di servizio ti scrive: "Ho fatto un molecolare e sono risultata positiva".
>
> .. Non potrai farti un tampone prima delle 8:00 dell'indomani quindi sai che ti aspettano almeno dieci ore di panico controllato. Qualcuna di meno, se riesci ad addormentarti. Cerchi di distrarti guardando la televisione, ma l'ennesimo thriller con Jason Statham, intervallato da pubblicità di ansiolitici (un conflitto di interessi che ti riprometti di studiare con più attenzione, se sopravvivi), non fa che aumentare la tua agitazione; così, spegni il televisore, ti prepari una tisana relax, <a href="http://chiplusplus.
>
> org">leggi un po' e poi cerchi di dormire.

Questo tipo di errori possono aspettare anni, prima di venire alla luce. Per esempio, un errore nella valutazione di una data in coincidenza con gli anni bisestili potrebbe aspettare quattro anni prima di manifestarsi; nel frattempo, il codice sarà stato distribuito agli utenti e chi lo ha scritto ne avrà perso memoria o potrebbe addirittura aver cambiato lavoro.
La correzione dell'errore della funzione `banner_testo`, se fatta per tempo, richiederebbe solo l'aggiunta di una condizione all'istruzione `if`, per verificare che il punto si trovi prima di un a capo:

```
if(c == PUNTO
&& letti >= N_CHAR_MIN
&& n_banner <= N_BANNER_MAX
&& testo.peek() == A_CAPO) {
    cout << endl
```

```
        << "<div id=\"banner-" << n_banner << "\">"
        << "</div>"
        << endl;
    n_banner++;
    letti = 0;
}
```

La stessa correzione, fatta dopo che il programma è andato in esercizio, potrebbe richiedere giorni, se non settimane, perché dovrà essere ripetuto tutto il processo di rilascio del sistema:

attività	ore/uomo
creazione di un ambiente di test	8
debug	2
correzione dell'errore	1
test funzionale	4
test di carico	8
test di sicurezza	8
collaudo	4
rilascio/distribuzione	1

Al costo di queste attività vanno ovviamente aggiunti i possibili danni derivanti dal mancato funzionamento del sistema, che potrebbero facilmente essere pari a un mese se non a un anno di stipendio del programmatore.
Devi pensare a tutto questo, quando scrivi codice, perché hai una responsabilità sia nei confronti del tuo datore di lavoro che degli utenti del sistema, che potrebbero essere anche i tuoi amici o i tuoi parenti.
Se lavori male per la Coca-Cola, puoi sempre pensare: "Chi se ne frega, io bevo Pepsi"; non è etico, ma almeno non è auto-lesionista. Se però lavori male per lo Stato, stai peggiorando la tua vita e di tutte le persone che conosci e questo, oltre a non essere etico, è anche stupido.

Gli **errori di analisi** sono una conseguenza dell'Annosa Dicotomia — e dell'inesperienza dell'analista, ovviamente.
Come tutti gli esseri senzienti, anche i clienti possono essere vittime dell'Annosa Dicotomia fra ciò che desiderano e ciò di cui realmente hanno bisogno. Un cliente che espone le sue esigenze è come un bambino che dice alla madre cosa vuole mangiare. Lo stimolo è reale — appetito o fame che sia —, ma il modo in cui lui vorrebbe placarlo non è necessariamente il più corretto; anzi: di solito è un desiderio indotto dal callido servitore dell'Entropia: il Marketing.
Il cliente ha diritto di comportarsi in questo modo: ciascuno di noi fa la stessa cosa quando entra in un negozio; come analista, però, non devi permettere che i tuoi giudizi siano influenzati dai desiderii del tuo cliente. In questo, l'Analisi è assimilabile a una disciplina mistica: così come il Buddha dà a ciascuna

persona con cui viene in contatto ciò di cui ha bisogno, tu devi annullare te stesso e *diventare* l'altro, pensare come lui, sentire le sue necessità. Da quello che dice e da come lo dice, devi riuscire a distinguere i suoi bisogni dai suoi desiderii, evitando che l'ottenimento di questi ultimi intralci il buon successo del progetto.
Ovviamente, dovrai anche evitare che i *tuoi* desiderii intralcino o rallentino il progetto. Così come influenza le opinioni del tuo cliente, il Marketing può influire su di te, spingendoti a scegliere una tecnologia di moda, ma inadatta allo scòpo che devi raggiungere. Fà attenzione che ciò non accada. Sorveglia costantemente le tue scelte; così come il Buon Programmatore, pensa sempre su più livelli e confrontati con altri colleghi o con il tuo superiore per essere certo di aver fatto le scelte giuste. Non vergognarti dei tuoi errori: sono i mattoni su cui edificherai la tua esperienza.
Sii anche preparato a dover fare degli errori. O prima o poi, capiterà che un cliente ti chieda di fare una scelta che tu ritieni sbagliata. Quando ciò accadrà, dovrai per prima cosa cercare di convincerlo dell'errore, spiegandogli perché la sua richiesta non sia corretta. Se non ci riesci, ripeti la tua opposizione, perché sia chiaro che la tua non è un'ipotesi, ma una certezza, però non insistere oltre: sia perché questo potrebbe creare degli attriti con il tuo interlocutore, sia perché alla lunga potresti scoprire che è lui ad avere ragione.
Se non riuscirai a convincere il cliente di un suo errore, dovrai portare a termine ciò che ti chiede, ma al contempo dovrai fare in modo che le conseguenze di quella scelta non possano essere attribuite a te in futuro. Dato che le conseguenze nefaste di un errore architetturale potrebbero manifestarsi dopo mesi o anche anni dalla fase di analisi, non limitarti a *dire* al cliente che si sbaglia, ma scrivilo, in modo che resti traccia della tua opposizione.
Il tuo peggior nemico, in questi casi, sarà il tuo orgoglio. Una volta, il Maestro Canaro mi disse:

> Quando ero un giovane *project-manager* e discutevo con i miei clienti, cercavo sempre di dimostrare loro che avevo ragione; ora cerco solo di far ciò che dev'essere fatto.

Parafrasando Iacopone da Todi, un programma per il debug può aiutarti a identificare il punto del tuo codice che genera un errore, ma devi prima capire quale sia la funzione da esaminare, perché fare il debug di tutto il codice di un programma, nei casi in cui questo sia possibile, sarebbe lungo ed estremamente frustrante.
Il modo in cui è stato scritto il codice lo renderà più o meno facile da verificare. Immagina che il problema sia la variabile `x`: se tutto il tuo codice ha la possibilità di modificarne il valore, potresti dover esaminare ogni singola funzione per verificare che non ne faccia un uso improprio. Al contrario, se la variabile `x` può essere modificata solo alcuni punti del codice, la tua sarà una ricerca più mirata e veloce. È per questo motivo, che nella lezione sulle funzioni iterative abbiamo diviso l'elaborazione dei dati dalla gestione dell'interfaccia utente: perché in

questo modo, a seconda del tipo di errore che dovesse presentarsi — di calcolo o di output — sapremo quale funzione andare a guardare.
Alcune caratteristiche del C++, come la tipizzazione forte e l'incapsulamento potranno esserti di aiuto in questo senso, ma non sempre saranno sufficienti a identificare il punto esatto in cui il tuo codice fa qualcosa di errato. In questi casi, dovrai procedere per tentativi, scomponendo il tuo programma in parti sempre più piccole, in modo da ridurre il numero di righe di codice da verificare. Un modo rapido per farlo è di mettere a commento tutte le chiamate nella funzione `main` ripristinandole poi a una a una, fino a che non individuerai quella in cui è contenuto l'errore:

```
int main(int argc, char** argv)
{
    ifstream testo;

    verifica_parametri(argc, argv);
    apri_file(testo, argv[2]);
/*
    elabora_file(testo);
    chiudi_file(testo);
*/
    return 0;
}
```

Se la *funzione che non funziona* è troppo complessa per farne un debug diretto, ripeterai il processo, mettendo a commento le sue chiamate fino a che la quantità di codice da esaminare sarà ragionevolmente poca.
Un altro modo in cui puoi semplificare la ricerca degli errori nel codice è l'aggiunta di messaggi che ti permettano di sapere quale operazione sta compiendo il programma:

```
int apri_file(ifstream& testo, const char* path)
{
#ifdef __LOG__
    log(LOG_DEBUG, 2, "Apro il file: ",  path);
#endif

    testo.open(path);
    return ERR_NONE;
}
```

La funzione `log` è quella che abbiamo visto nella lezione sulle funzioni con parametri variabili e ci permette di conoscere il nome del file che viene aperto durante l'esecuzione del programma.
Queste funzioni di tracciatura sono utili nella fase di debug, ma rallentano l'esecuzione del programma perché richiedono l'accesso a un dispositivo esterno, sia esso lo schermo del computer o un file sul disco rigido. Per questo motivo, è bene avere la possibilità di disabilitarle nella versione definitiva del programma.

In questo caso, l'abbiamo fatto inserendo la chiamata in una direttiva `ifdef` del precompilatore, in modo che venga inserita nel codice solo se è definita la costante `__LOG__`. Dato che si tratta di una costante che non viene utilizzata dal codice, ma che serve solo per modificare il modo in cui è compilato il programma, possiamo definire `__LOG__` direttamente nella linea di comando del compilatore:

```
> g++ sorgente.cpp -D __LOG__ -o eseguibile
```

Quando l'errore si manifesterà — di solito pochi minuti prima che tu debba smettere di lavorare per uscire o fare qualcos'altro — e tu dovrai identificarne la causa, il primo problema che avrai sarà di riuscire a riprodurre le condizioni in cui si manifesta. Come abbiamo visto poco fa, se l'errore dipende dai dati in input, per identificare il problema, dovrai capire quali sono i dati che lo generano; qualche volta sarà facile, ma in altri casi potrà rivelarsi estremamente complesso.
Diversi anni or sono, il Maestro Canaro dovette registrarsi su un sito Web che gli chiese anche la sua data di nascita — che, come sai, fu il 29 Febbraio del 1964 . La maschera di inserimento nuovo utente non gli diede problemi, ma la maschera di modifica dati, evidentemente scritta da un programmatore meno esperto, non gli permise di aggiornarli perché, a suo dire, la data di nascita era sbagliata. Ciò vuol dire che il sistema utilizzava due funzioni distinte per il controllo della data di nascita, una nella funzione di inserimento e un'altra nella funzione di modifica, e che almeno la funzione utilizzata in modifica non era una funzione standard, ma codice scritto *ad-hoc*.
Entrambe queste scelte sono errori: a una determinata azione sui dati deve corrispondere una singola funzione. Fare la stessa operazione con parti di codice distinte è sbagliato; sia perché aumenta la probabilità di commettere degli errori, sia perché rallenta i tempi di identificazione dell'errore in fase di debug. È sbagliato anche riscrivere delle funzioni che già esistono: D. J. Bernstein lo fece, con le funzioni di I/O di qmail, ma la sua fu una precisa scelta architetturale perché voleva delle funzioni che fossero migliori e più sicure delle funzioni della libreria standard.
A questi due errori di programmazione — inammissibili, in un sito che gestisca transazioni economiche — si aggiunge una profonda sciatteria della fase di debug del codice, perché la corretta gestione dei casi particolari, come gli anni bisestili, va sempre verificata. Quando verifichi il funzionamento di un programma, non puoi limitarti a controllare che faccia ciò che deve fare, ma devi anche assicurarti che non faccia ciò che non deve fare. In particolare, devi verificare che si comporti correttamente se:

- gli fornisci i dati di input corretti;
- non gli fornisci alcun dato;
- gli fornisci dati errati;
- gli fornisci dati in eccesso.

Quindi, se l'input è una data, dovrai verificare che il tuo sistema gestisca correttamente sia il valore `29-02-1964` che il valore `29-02-1965`; se l'input è una stringa di testo, dovrai accertarti che il sistema gestisca correttamente anche il caso in cui riceva più caratteri del previsto e che elimini eventuali caratteri di spazio all'inizio o alla fine del testo, a meno che questo non sia un requisito funzionale.

In questo programma, una piccola cosa non è stata fatta come si dovrebbe e ne è derivato un errore:

```
/**
 * @file src/debug-gestione-errori.cpp
 * Modalità di gestione degli errori.
 */

#include <iostream>
#include <fstream>
#include <exception>

using namespace std;

/** Codici e stringhe di errrore */
#define LOG_DEBUG         1
#define LOG_AVVISO        2
#define LOG_ERRORE        3
#define ERR_NONE          0
#define ERR_FILE_NONE   -10
#define ERR_FILE_OPEN   -20
#define ERR_FILE_READ   -30
#define S_DEBUG         "DEBUG"
#define S_AVVISO        "AVVISO"
#define S_ERRORE        "ERRORE"
#define S_ERR_FILE_NONE "Definire un file di input"
#define S_ERR_FILE_OPEN "Impossibile aprire il file di input"
#define S_ERR_FILE_READ "Impossibile leggere il file di input"

/**
*   Definisce una classe derivata da exception
*   per la gestione degli errori.
*/
class Eccezione: public exception
{
private:
    int         _codice;
    const char* _errore;
```

```
public:

    /** Costruttore */
    Eccezione(int codice, const char* errore)
    : _codice(codice), _errore(errore) {
    }

    /** Funzione virtuale pura: va ridefinita */
    virtual const char* what() const throw() {
        return _errore;
    }

    /** Funzioni di interfaccia */
    int getCodice() { return _codice; }
    const char* getErrore() { return _errore; }

    /** Ridefinizione dell'operatore di output */
    friend ostream& operator<< (ostream& os, Eccezione e){
        os << e._codice << ": " << e._errore << endl;
        return os;
    }
};

/** Funzione di gestione degli errori */
void errore(int codice, bool exit = true)
{
    const char* errore = NULL;

    /**
    *   Identifica la stringa di errore corrispondente
    *   al codice di errore ricevuto.
    */
    switch(codice) {
        case ERR_FILE_NONE: errore = S_ERR_FILE_NONE; break;
        case ERR_FILE_OPEN: errore = S_ERR_FILE_OPEN; break;
        case ERR_FILE_READ: errore = S_ERR_FILE_READ; break;
    }

    /** Se ne trova una, lancia un'eccezione */
    if(errore != NULL){
        Eccezione e(codice, errore);
        throw e;
    }
}

/**
```

```
*   Questa è la funzione di log che abbiamo visto
*   nella lezione sulle funzioni.
*/
void log(int livello, int n_parametri, ...)
{
    /** Definisce il livello del messaggio */
    const char* s_livello;
    switch(livello) {
        case LOG_DEBUG:  s_livello = S_DEBUG ; break;
        case LOG_AVVISO: s_livello = S_AVVISO; break;
        default:         s_livello = S_ERRORE; break;
    }

    /** Scrive il testo del messaggio */
    cerr << '[' << s_livello << "] ";

    /** Scrive i parametri della lista */
    va_list lista_parametri;
    va_start(lista_parametri, n_parametri);
    for(int p = 1; p <= n_parametri; p++) {
        cerr << va_arg(lista_parametri, char*) ;
    }
    va_end(lista_parametri);

    cerr << endl;
}

/** Verifica la presenza dei parametri di avvio */
int verifica_parametri(int argc, char** argv)
{
    return (argc < 2) ? ERR_FILE_NONE : ERR_NONE;
}

/** Apre il file in lettura */
int apri_file(ifstream& testo, const char* path)
{
/**
*   Questo codice viene compilato solo se
*   è definita la macro __LOG__
*/
#ifdef __LOG__
    log(LOG_DEBUG, 2, "Apro il file: ",  path);
#endif

    testo.open(path);
    return ERR_NONE;
```

```
}

/** Legge il file di input e lo scrive a video */
int elabora_file(ifstream& testo)
{
    int  letti = 0;
    char c = 0;
    while ((c = testo.get()) != EOF) {
        letti++;
        cout << c;
    }
    return letti;
}

/** Chiude il file di input */
void chiudi_file(ifstream& testo)
{
    testo.close();
}

int main(int argc, char** argv)
{
    ifstream testo;

    try {

        int esito = ERR_NONE;

        /**
        *   Verifica che ci sia il nome del file di input
        *   e gestisce eventuali errori.
        */
        esito = verifica_parametri(argc, argv);
        errore(esito);

        /**
        *   Imposta la exception mask dello stream per fare
        *   sì che un errore di I/O generi un'eccezione,
        *   poi apre il file in lettura.
        *   Usa un blocco try/catch per intercettare una
        *   eventuale eccezione e gestirla in maniera
        *   omogenea al resto del codice.
        *   Non può aggiungere una catch in fondo perché
        *   l'eccezione non dà informazioni sulla causa
        *   dell'errore che l'ha generata.
        */
```

```
        try {
            testo.exceptions ( std::ifstream::badbit
                             | std::ifstream::failbit );
            esito = apri_file(testo, argv[2]);
        } catch(ifstream::failure e) {
            errore(ERR_FILE_OPEN);
        }

        /**
        *   Elabora il testo e gestisce eventuali errori.
        *   Dato che la funzione torna il numero di
        *   caratteri letti, deve chiamare errore solo
        *   se non ce ne sono.
        */
        testo.exceptions ( std::ifstream::goodbit);
        if(elabora_file(testo) == 0) {
            errore(ERR_FILE_READ);
        }


        /** Chiude il file di input */
        chiudi_file(testo);

    } catch (Eccezione e) {
        cerr << e;
        exit(e.getCodice());
    }

    return 0;
}
```

In ossequio a quanto abbiamo detto poco fa, per verificare il funzionamento di questo programma dovremo fare almeno quattro prove:

```
# dati corretti
src/out/esempio src/cpp/debug-testo-1.txt

# nessun dato
src/out/esempio

# dati errati
src/out/esempio src/cpp/file-inesistente

# dati in eccesso
src/out/esempio src/cpp/debug-testo-1.txt abcdefghilmenopqrstuvz
```

Dobbiamo poi verificare che tutte le condizioni di errore siano gestite corretta-

mente. Nel nostro caso, gli errori previsti sono:

```
#define ERR_FILE_NONE   -10
#define ERR_FILE_OPEN   -20
#define ERR_FILE_READ   -30
```

I primi due errori sono verificati dalle prove standard; il terzo caso lo possiamo verificare passando al programma un file vuoto:

```
src/out/esempio src/cpp/debug-vuoto.txt
```

Sfortunatamente, però, se compili ed esegui questo codice con i dati corretti, ottieni un errore, anche se il file esiste:

```
> g++ src/cpp/debug-gestione-errori.cpp -o src/out/esempio
> src/out/esempio src/cpp/debug-testo-1.txt
-20: Impossibile aprire il file di input
> ls src/cpp/debug-testo-1.txt
src/cpp/debug-testo-1.txt
```

Se ri-compili il programma definendo la macro `__LOG__` per verificare quale sia il file che il programma sta aprendo:

```
#ifdef __LOG__
    log(LOG_DEBUG, 2, "Apro il file: ",  path);
#endif
```

quando esegui i programma, ottieni un nuovo errore:

```
> g++ src/cpp/debug-gestione-errori.cpp -D __LOG__ -o src/out/esempio
> src/out/esempio src/cpp/debug-testo-1.txt
[DEBUG] Apro il file: zsh: segmentation fault  src/out/esempio src/cpp/debug-testo-1.txt
```

Questo non è il comportamento atteso dalla funzione, ma ci permette comunque di capire quale possa essere il problema. L'errore: `segmentation fault` vuol dire che il programma sta cercando di accedere a un'area di memoria che non gli appartiene. L'area di memoria in questione è quella associata al parametro `path`, che a sua volta è stato inizializzato con il valore della variabile `argv[2]`:

```
esito = apri_file(testo, argv[2]);
```

Il *bug* è l'indice `2` nell'array `argv`. Come certamente avrai notato, il codice di questo programma è una rielaborazione del codice della lezione sugli stream, che doveva gestire tre parametri da riga di comando. Stavolta, però, la stringa di chiamata del programma ha solo due valori: il path del programma e il nome del file di input:

```
> src/out/esempio src/cpp/debug-testo-1.txt
```

Riutilizzare il codice è una cosa buona; scordarsi di modificare il valore dell'indice dell'array `argv` è una cosa sbagliata, perché l'indirizzo di memoria puntato da `argv[2]`, adesso, non appartiene al programma: non possiamo utilizzarlo come path per una funzione `open` e non possiamo stamparlo a video.

Se correggiamo l'indice, il programma gestisce correttamente tutte le condizioni d'uso:

```
> g++ src/cpp/debug-gestione-errori.cpp \
    -D __LOG__ \
    -o src/out/esempio
> src/out/esempio
-10: Definire un file di input

> src/out/esempio src/cpp/file-inesistente
[DEBUG] Apro il file: src/cpp/file-inesistente
-20: Impossibile aprire il file di input

> src/out/esempio src/cpp/debug-vuoto.txt
[DEBUG] Apro il file: src/cpp/debug-vuoto.txt
-30: Impossibile leggere il file di input

> src/out/esempio src/cpp/debug-testo-1.txt
[DEBUG] Apro il file: src/cpp/debug-testo-1.txt
Essere un ossessivo-compulsivo con una leggera tendenza ...
```

Questo errore di distrazione è stato facilitato dall'utilizzo di una costante numerica per la definizione dell'indice dell'array. Scrivere direttamente un numero o una stringa nel codice è sicuramente più rapido e allettante che definire delle costanti per il precompilatore:

```
#define PARAM_PATH 1
...
esito = apri_file(testo, argv[PARAM_PATH]);
```

ma, sul lungo periodo, è controproducente perché rende il codice più complesso da leggere e da modificare.
Più complesso da leggere perché le costanti aiutano a capire cosa faccia il codice. Se leggi l'istruzione:

```
return (argc < 2) ? ERR_FILE_NONE : ERR_NONE;
```

puoi capire cosa faccia anche se non conosci il codice. Se invece leggi la stessa istruzione, ma senza le costanti:

```
return (argc < 2) ? -10 : 0;
```

per capire cosa faccia dovrai andare a leggere la documentazione del programma, posto che ce ne sia una.
Più complesso da modificare perché l'utilizzo di costanti al posto di valori *hard-coded* permette di cambiare il valore di una costante agendo in un solo punto:

```
#define ERR_NONE  1
```

Se non avessimo usato una costante, per ottenere lo stesso risultato avremmo dovuto modificare tre istruzioni distinte:

```
return (argc < 2) ? ERR_FILE_NONE : ERR_NONE;

...

return ERR_NONE;

...

int esito = ERR_NONE;
```

In un programma più complesso del nostro esempio, le modifiche sarebbero state sicuramente di più e più difficili da identificare; inoltre, se ce ne fossimo dimenticata una (probabile), avremmo introdotto un errore nel sistema.
Le costanti *hard-coded* possono essere utilizzate solo nella prima fase dello sviluppo del programma, quando non sei ancora sicuro che la strada che hai scelto sia quella giusta. In questa fase è ammissibile che tu faccia delle prove inserendo dei valori direttamente nel codice, ma quando l'algoritmo sarà ragionevolmente stabile, dovrai convertire tutti i valori in costanti.

> Un sistema è una casa che, subito dopo costruita e adornata, ha bisogno (soggetta com'è all'azione corroditrice degli elementi) di un lavorio più o meno energico, ma assiduo, di manutenzione, e che a un certo momento non giova più restaurare e puntellare, e bisogna gettare a terra e ricostruire dalle fondamenta. Ma con siffatta differenza capitale: che, nell'opera del pensiero, la casa perpetuamente nuova e sostenuta perpetuamente dall'antica, la quale, quasi per opera magica, perdura in essa.

O prima o poi, la tua vita andrà in errore, come il software.
Non importa quanto tu sia stato prudente o quale sia il tuo *Karma*: a un certo punto la terra sotto i tuoi piedi comincerà a franare e tu cadrai giù, lungo la montagna che stavi scalando, ritrovandoti al punto di partenza. Quello che farai in quel momento deciderà del resto della tua vita e ti farà capire che tipo di uomo sei — o che donna, visto che queste cose non succedono solo ai maschietti.
Comincia a prepararti da adesso a quel momento, perché, quando avverrà (*quando*, non: *se*), probabilmente sarai solo e prendere delle decisioni sarà molto difficile perché avrai perso ogni fiducia in te stesso. Se ti rompi una gamba, o un braccio, il tuo cervello ti può dire se stanno guarendo o peggiorando, ma se batti la testa non è facile capire come stai, perché l'organo in esame e l'organo esaminatore coincidono. Allo stesso modo, se non ti fidi di te stesso, è difficile capire se le scelte che fai sono corrette. La paura o la prudenza potrebbero spingerti a non fare la scelta giusta, quindi, per prima cosa, dovrai fare il *debug* della tua vita per capire se e in quale misura devi biasimarti per ciò che è avvenuto; fatto ciò, dovrai identificare i tuoi errori e trovare un modo per non ripeterli.
L'approccio più comune è di guardarsi indietro e cercare di capire quali siano state le proprie colpe, considerando queste degli eventi isolati in un'esistenza

fatta prevalentemente di scelte corrette. Dato però che ciascuno di noi tende — più o meno inconsciamente —, a cercare cause esogene alle sue sventure inventandosi complotti o trasferendo le proprie responsabilità a terzi, la cosa migliore, in questi casi, è di adottare l'atteggiamento opposto e partire dal presupposto che *tutto* ciò che è successo di male nella tua vita sia una tua colpa, per poi individuare i casi in cui ciò che è successo, in effetti, non è dipeso da te. Questo approccio *bottom-down* ha due pregi: il primo è che, analizzando gli eventi passati potresti scoprire che alcune colpe che ti attribuivi non erano reali; il secondo è che sarà più difficile mentire a te stesso. Ciascuno di noi ha una parte di responsabilità in ciò che gli succede, anche negli eventi che non genera direttamente. Assumersi a priori la colpa di tutto il male che ci è successo rende più difficile mentirci e ci permette di identificare tutte le nostre colpe, per piccole che siano.
Attenzione, però: non devi pensare alle tue colpe come se fossi un inquisitore del tredicesimo Secolo, ma come se fossero dei *bug* nel programma della tua vita. Qualcuna genererà degli errori, altre solo dei *warning* e il tuo dovere è quello di identificarne il più possibile, per poi cercare di correggerle per migliorare il funzionamento del sistema.

> Il cambiamento inizia quando si intraprende un nuovo sentiero , anche se questo sentiero non è che una traccia lasciata da una capra assetata che ha trovato una sorgente.

Contrariamente a quello che avviene con il software, non sempre è possibile correggere i *bug* della nostra esistenza. Si può modificare un aspetto deteriore del nostro carattere, ma non è detto che sia possibile rimediare ai danni che questo ha causato a noi o a terzi. Per fare un paragone con il debug del software, i difetti caratteriali sono errori di compilazione, mentre gli effetti dei nostri sbagli sono errori di esecuzione; i primi li possiamo correggere, gli errori di esecuzione, no: ormai è andata. Quello che possiamo e che dobbiamo fare, però, è di pentircene sinceramente, ovvero riconoscerli come errori, in modo da evitare di ripeterli nei prossimi cicli di esistenza.
Anche se non possiamo cancellare gli effetti di un nostro errore, possiamo comunque chiedere perdòno a coloro i quali abbiamo arrecato danno. Non basterà chiedere scusa: si chiede scusa quando ciò che hai fatto non dipende da te, come quando qualcuno di urta e tu versi il tuo vino sul vestito del vicino; se invece il vino glielo hai tirato addosso intenzionalmente, dovrai chiedere *perdòno*, cercare per quanto possibile di riparare all'errore fatto e non commetterlo mai più, né con lui (o lei) né con altri.
Non sottovalutare il potere terapeutico del perdòno, anche se ci saranno dei casi in cui non lo otterrai. Se è vero che non si può cambiare il passato per qualcuno che abbiamo ferito, è altrettanto vero che si può provare a compensare il danno fatto rendendogli migliore il presente o il futuro. Le persone a cui facciamo del male sono spesso quelle a cui siamo più legati; far sapere loro che che non li abbiamo dimenticati e che ciò che è successo ci addolora, può servire a ricucire delle ferite; dall'una e dall'altra parte.

Quando ripenserai a ciò che ti è avvenuto in passato, come un buon analista, dovrai cercare di immedesimarti nelle persone con cui hai avuto a che fare, capire le loro ragioni al di là di eventuali rancori o recriminazioni. Per fare ciò, dovrai tenere a mente alcuni fattori che influenzano il comportamento di tutti noi. Il primo, ovviamente, è l'influenza dell'Annosa Dicotomia, che, per mano del suo lacché il Marketing, spinge le persone a soddisfare i proprii desiderii invece che i proprii bisogni, creando degli schemi di valori fallaci e spingendoli a dimenticare che esistono per tutti la vecchiaia e la morte.
Il secondo fattore da considerare è la natura umana; ricorda:

> Non cercare di spiegare con la malizia quello che può essere spiegato con la stupidità.

O, per dirla con De Santillana:

> C'è una buona regola per gli storici che ho sempre tenuto presente quando lavoravo al mio *Galileo*: mai sottovalutare il potere della stoltezza e dell'insensatezza nelle vicende umane.

Per derimere le questioni relative ai rapporti di coppia, invece, l'approccio più sicuro è quello antropologico. Come diceva il Maestro Canaro:

> Cento anni di femminismo non possono battere centomila anni di evoluzione.

Secondo lui, qualsiasi comportamento anomalo nell'ambito di una coppia può essere spiegato tenendo a mente tre principii:

1. le donne sono incubatrici parlanti;

2. gli uomini sono dispenser di sperma;

3. ogni eccesso nasconde un eccesso di natura opposta e pari entità.

Parafrasando quel senza Dio di Dawkins, noi siamo l'*hard-disk* dei nostri geni, la memoria di massa che garantisce loro una persistenza. Il nostro software può variare, ma il firmware che definisce il nostro comportamento a basso livello è immutato da migliaia di anni e ci spinge a fare ciò per cui siamo stati creati, ovvero riprodurci.
Né le sovrastrutture culturali che abbiamo inventato, né gli idoli ai cui piedi ci prostriamo e nemmeno l'Annosa Dicotomia possono modificare la nostra ROM. Ignorare o, peggio, ribellarsi a questo stato di fatto è il primo passo verso la rovina o l'infelicità o entrambe le cose.

Ecco: questo è tutto. Ricorda però: ciò che ti ho insegnato non è un punto di arrivo, ma un punto di partenza. La fine del cammino che abbiamo percorso insieme coincide con l'inizio del cammino che percorrerai da solo. Da questo

momento in poi tu hai il dovere di diffondere le idee che ti ho trasmesso e, allo stesso tempo, di metterle costantemente alla prova per emendarle dai molti errori che certamente avrò commesso, così come il Maestro Canaro prima di me. Così come agli antichi Cristiani era vietato adorare gli idoli, io ti vieto di adorare le parole. Scrivi il tuo libro, se lo desideri, ma che non sia un libro stupido; fa' in modo che lo si possa correggere facilmente, se necessario, in modo che ciò che afferma sia sempre il punto più vicino alla verità che tu possa raggiungere, perché l'oggetto dei tuoi sforzi dev'essere sempre la verità, non la tradizione.
Noi viviamo prevalentemente sulla terraferma e riteniamo perciò che la normalità sia questa. Se però si trascorre un lungo periodo di tempo su una nave o in un'isola, si ha modo di capire come la normalità sia l'acqua e la terra sia solo un'eccezione. Similmente, noi diamo grande importanza al ciclo di nascita, riproduzione e morte che chiamiamo "vita" e in essa vediamo il fine ultimo dell'Universo, dimenticandoci che la vita è solo un caso particolare di esistenza e che un universo di sassi sarebbe comunque prodigioso.
Ciascuna forma di vita consociata, per poter sopravvivere, richiede la legittimazione di alcune follie biologiche. Il rispetto di queste follie, essendo in-naturale, richiede l'istituzione di un livello superiore di costrizione, ovvero un corpo di regole formali ancora più distante del precedente dal comportamento biologico degli esseri viventi. Con il passare del tempo, molte di queste convenzioni nate in seguito a esigenze contingenti, finiscono inevitabilmente per contrastare con il senso comune di giustizia e devono essere o abolite o modificate, solo che ciascuna modifica o adattamento invece di renderle più "giuste", le allontana ulteriormente dal loro scòpo iniziale e le rende soggette a interpretazioni errate o addirittura opposte a quelle che era il loro fine primario.
Sant'Agostino ha detto la stessa cosa, ma molto meglio di quanto stia facendo io adesso.
Si vede che era sobrio.

Epilogo

Dopo di me, il reboot

Fu uno dei confratelli ad avvisarmi. Aprì la porta della mia cella senza nemmeno bussare, una sagoma nera contro la luce del corridoio alle sue spalle.
«Il vecchio si è sentito male, lo hanno portato in infermeria,» disse. «È meglio che ti sbrighi se lo vuoi salutare, non credo che gli resta molto tempo.»
Avrei dovuto essergli grato, per aver pensato ad avvisarmi, invece lo odiai, sia per la notizia che per l'errore nel congiuntivo. Mi alzai e mi vestii al buio, pensando a quanto sono astuti i Benedettini che dormono vestiti e che non devono inquinare la drammaticità di momenti simili con pensieri come: "Dove ho messo le mutande?".

Appena ebbi qualcosa addosso, uscii di corsa e, a piedi nudi, scesi in infermeria. Indugiai prima di aprire la porta così come avevo indugiato davanti alla porta della sua cella la prima volta che ero andato a cercarlo. Quando alla fine mi feci coraggio e l'aprii, lo vidi sdraiato su uno dei letti in fondo alla stanza. Era appena un'increspatura sul piano della coperta, ma la coperta — notai con un certo sollievo —, si muoveva leggermente, seguendo il flusso del suo respiro. Lui si accorse della mia presenza (o forse sapeva che sarei arrivato) e, con un filo di voce disse:
«Toh! guarda chi si vede: Nino lo studente..»
Stava morendo, probabilmente era l'ultima volta che ci vedevamo e mi salutava con una battuta di un film di Nino D'Angelo. Odiaii anche lui, ma per poco; giusto il tempo di capire perché l'aveva fatto.
«Sei contento, ora?» mi chiese il Maestro.
«Di cosa?» chiesi, e aggiunsi: «No, non sono contento. Perché dovrei essere contento?» Il Maestro fece due respiri prima di rispondere.
«Hai il tuo libro, no? Ti ho insegnato tutto quello che so; non servo più a nulla.»
Non capivo: voleva dire che stava morendo a causa mia?
«Non capisco,» dissi. «Vuol dire che sta morendo a causa mia?»
(Non so se ci avete fatto caso, ma quando muore qualcuno che amiamo, i pensieri e le frasi tendono a diventare meno letterarii e più elementari.)
«Tutto quello che ti ho insegnato, pensavi che scherzassi? Siamo qui per uno scopo; quando non serviamo più, possiamo essere rimossi.» Tossì, poi riprese: «L'operatore `delete`, ricordi? Bisogna liberare la memoria, altrimenti il programma non avrà più spazio per girare.»
Lo avevo ucciso io? Davvero? Sentii il cuore pulsarmi nelle orecchie e guardai con cupidigia uno sfigmo-manometro poggiato su uno dei tavoli dell'infermeria; chissà quanto avevo di pressione.
«Il Cielo non fa favoritismi: per lui, siamo tutti preziosi e inutili allo stesso tempo, come la Regina in una partita di Scacchi. È il pezzo più importante, dopo il Re, ma un buon giocatore non ha problemi a sacrificarlo, se questo gli permette di vincere la partita.»
Mi fece cenno di dargli un po' d'acqua; bevve, poi riprese a parlare.
«O forse no. Ti è mai passato per la testa, che tutta questa teoria potrebbe essere sbagliata? Che potrei essere davvero pazzo, come dicono i tuoi confratelli? Wittgenstein a trent'anni, definì una filosofia che egli stesso rinnegò dieci anni dopo, a favore di un nuovo credo. Se fosse vissuto per altri dieci anni, avrebbe cambiato ancora idea? La storia dell'Umanità è costellata di idee bizzarre, ciascuna con il suo bravo seguito di fedeli; cosa ti fa pensare, che il C'hi++ sia diverso? Il fatto che possa dare una risposta razionale e coerente ad alcuni fenomeni che altrimenti sarebbero senza spiegazione non vuol dire necessariamente che corrisponda a verità. Sì, certo: i suoi assiomi sembrano trovare conferma nella realtà, ma questo cosa conta? Potremmo essere noi, che non ci accorgiamo degli errori. Abbiamo creduto per centinaja di anni che il Sole girasse intorno alla terra e anche quella era un'ipotesi confortata dai fatti, almeno apparentemente.»
Sorrise del mio sguardo perplesso, poi riprese a parlare;

«Sta' tranquillo: non sono pazzo, ma tu non pensare che quello che ti ho insegnato sia scienza. C'hi++ non è né scienza né religione, anche se ogni tanto finge di essere o l'una o l'altra cosa. Cerca la Verità, come la scienza e prova dare un significato alla nostra esistenza, come la religione, ma non può dimostrare ciò che afferma, come la scienza, e non vuole che tu lo accetti per forza, come la religione. Lo scòpo del C'hi++ è aiutarci a vivere meglio, quindi, quello che ti devi chiedere non è se le tesi dello Spazionismo siano corrette, ma se quello che ti ho insegnato ti rende la vita migliore o no.»
Bevve un altro sorso di acqua, poi, con gli occhi chiusi, disse:
«Il tuo maestro sta morendo e tu sei visibilmente addolorato, quindi è un ottimo momento per verificarlo. Va' fuori, guarda in alto e chiediti se quello che ti ho insegnato rende questo momento meno difficile da affrontare.»
Replicai che non mi sembrava il caso: pioveva ed ero a piedi nudi, ma era solo parte del problema: in realtà non volevo lasciarlo perché mi sembrava che le sue condizioni stessero rapidamente peggiorando.
«È l'ultima cosa che ti chiedo. Davvero me la vuoi negare?»
Mi arresi: uscii fuori, sotto la pioggia, e guardai in alto, come mi era stato ordinato di fare. La luna e le stelle erano nascoste dalle nuvole e comunque la pioggia colpiva i miei occhi impedendomi di vedere con chiarezza. Ciò non ostante, o forse proprio per questo motivo, non abbassai lo sguardo e lasciai che le gocce di pioggia si confondessero con le lacrime. Fu così che capii. Quelle gocce di acqua che adesso erano pioggia, erano state, prima, vapore acqueo e mare e fiume e prima ancora — per quello che ne potevo sapere — sangue, sudore, piscio, ghiaccio, vino, muscoli, piante, saliva, fango, calce o cemento. A qualunque cosa si fossero legate, nella loro esistenza precedente, adesso erano di nuovo acqua, ed erano pronte per un altro ciclo di vita. Il calore del sole le aveva portate in alto, ora la gravità le riportava in basso, verso il mare.
Tornai nella cella del Maestro per dirgli che avevo capito, ma la coperta non si muoveva più.
"Non importa", pensai. "Glielo dirò la prossima volta".

Questa è la poesia funeraria del Maestro. Una poesia funeraria "ready-made", come la definiva lui, perché il testo è quello della poesia funeraria del monaco Zen Gesshu Soko. Il Maestro la tradusse in Tedesco quando scoprì che il verbo *zu Treffen* può significare sia: *incontrarsi* che: *fare centro.*

> Einatmen, Ausatmen,
> Vorwärts gehen, Rückwärts gehen,
> Leben, sterben, kommen, gehen.
> Wie zwei Pfeile, die sich im Flug treffen.
> Mitten im Nichts,
> Eine Straße, die direkt zu meinem wirklichen Zuhause führt.

Ringraziamenti

A mia madre, per la favola del Cavallo Fuggito e per aver piantato il seme da cui è nato tutto questo.
A mio padre, per avermi insegnato a distinguere uno scrittore onesto da uno disonesto, anche quando lo scrittore sono io.
A Wayne, per avermi regalato il libro.
A Claudio, per avermi insegnato a programmare i computer e per Coomaraw-sami.
A Manuela, per Musashi.
A Luciano, per aver detto che il testo non si capiva.
Al Cane Lele, per avermi aiutato a capire.

Note

Le note sono la parte più importante di un libro. William Hazlitt, nel primo capitolo de: *L'ignoranza delle persone colte*, sostiene che coloro i quali leggono troppi libri non hanno idee proprie e quindi sono costretti a chiedere in prestito la saggezza altrui; allo stesso tempo, con encomiabile coerenza, in circa otto pagine di testo, fa una dozzina di citazioni: dal Vangelo a Milton, da Shakespeare a Wordsworth.
Vorrei perciò che tu soffermassi la tua mente sul motivo che ti ha spinto a riportare qui dei brani di opere altrui. Se è un modo di dire: "Vedi? Anche lui la pensava come me!", è sbagliato; se mai, devi affermare: "Vedi? anche io la penso come lui!". Come ti ho detto in precedenza, la Verità è una; puoi ritrarla a olio, a tempera, a carboncino o al tratto, ma il soggetto della tua opera sarà sempre lo stesso; se non lo è, non è di Lei, che stai parlando.
René Guénon disse:

> Sembra perfino che ai filosofi importi assai più porre dei "problemi", siano pur essi artificiali e illusori, che non risolverli: il che costituisce un aspetto del bisogno disordinato della ricerca per la ricerca, cioè dell'agitazione più vana, nell'ordine mentale non meno che nell'ordine corporeo. A questi stessi filosofi interessa altresì legare il loro nome ad un "sistema", cioè ad un insieme di teorie strettamente delimitato, che sia il loro e non significhi altro che l'opera loro. Donde il desiderio di esser originali ad ogni costo, perfino se la verità dovesse venir sacrificata a siffatta originalità. Per la reputazione di

> un filosofo vale assai più inventare un errore nuovo che ripetere una verità già espressa da altri. Questa forma di individualismo, cui si devono tanti "sistemi" contradittori fra loro, se non pure in se stessi, si ritrova peraltro in egual misura fra gli scienziati e gli artisti moderni. Ma è forse tra i filosofi che l'anarchia intellettuale, che ne è la conseguenza inevitabile, spicca più nettamente.

Questo *bisogno* di distinguersi, questa *agitazione* sono chiari sintomi di un asservimento all'Entropia — ammissibile per uno *kshatriya*, ma inaccettabile per un *brâhmana*.
Così come i boschi di Faulkner non erano né del Maggiore De Spain, né di Thomas Sutpen, da cui li aveva comprati, né del capo Chickasaw Ikkemotubbe che glieli aveva venduti, Linux non appartiene né ai suoi sviluppatori, né a Linus Torvalds e nemmeno ad Andrew Tanenbaum. Allo stesso modo, non si può dire che il C'hi++ sia tuo, perché lo hai descritto nel tuo libro, né mio, che te l'ho insegnato, né del Maestro Canaro che a sua volta lo insegnò a me. Come avviene con la produzione del software, ciascuno di noi ha descritto a suo modo una stessa modella, aggiungendo qualcosa, correggendo qualcosa e, inevitabilmente, introducendo nuovi errori, che dovranno essere corretti da chi verrà dopo di noi. Non c'è niente di male, in tutto ciò. Non è degli errori, che devi avere paura, ma della stasi.

Il buon programmatore

1. Piera Scarabelli, Massimo Vinti, *Bhagavad Gita: Con un commento del testo basato sul Gītā Bhāṣya di Rāmānuja*. Mimesis Edizioni, Milano, 2017

Il C++

1. Bjarne Stroustrup, gotw.ca/publications/c_family_interview.htm

Tipi di dato

1. Agostino, *Confessioni* (VII, 11)
2. *"Love thou thy dream*
   all base love scorning,
   love thou the wind
   and here take warning
   that dreams alone can truly be,
   for 'tis in dream I come to thee"
   Ezra Pound, *Song*

Struttura dei programmi C++

1. *"Il Brahman ha due aspetti, il tempo e il Non-Tempo. Quello che è prima del sole è il Non-Tempo, senza parti"*

Maitri Upanishad (VI, 15)
2. Piera Scarabelli, Massimo Vinti, *op.cit.*
3. Richard Dawkins, *L'Orologiaio Cieco*, Mondadori, Milano, 2016

Gli operatori

1. Farid al-Din 'Attar, *Il verbo degli uccelli (Mantiq al-Tayr)*, a cura di Carlo Saccone, Centro Essad Bey-Amazon IP, Seattle 2016 (quarta ed.)
2. *Questi esempi indicano un'oggettività nella verità e nella falsità: ciò che è vero (o falso) è uno stato dell'organismo, ma in genere è vero (o falso) a motivo di avvenimenti esterni all'organismo.*
   Bertand Russell *Storia della filosofia occidentale*, Longanesi, Milano, 2014
3. *Es. 32.27*
4. Dati SIPRI www.sipri.org.
5. Definire un principio: *self-evident* equivale a dichiarare un dogma: una convinzione che appartiene a chi scrive, ma non si applica necessariamente a tutti gli esseri viventi.
6. Franklin credeva in Dio e in Gesù come filosofo; in una lettera al Presidente di Yale, che gli chiedeva quali fossero le sue idee religiose, scrisse: *I believe in one God, Creator of the Universe. That He governs it by His Providence. That he ought to be worshipped. That the most acceptable Service we render to him, is doing Good to his other Children. That the Soul of Man is immortal, and will be treated with Justice in another Life respecting its Conduct in this ... As for Jesus of Nazareth ... I think the system of Morals and Religion as he left them to us, the best the World ever saw ... but I have ... some Doubts to his Divinity; though' it is a Question I do not dogmatism upon, having never studied it, and think it is needless to busy myself with it now, where I expect soon an Opportunity of knowing the Truth with less Trouble.* Jefferson aveva idee abbastanza simili; Sherman era un Puritano, Livingston un Episcopale e John Adams si definiva: "a church going animal".
7. Henry Wiencek, *The Dark Side of Thomas Jefferson*, Smithsonian Magazine, Ottobre 2012. (www.smithsonianmag.com/history/the-dark-side-of-thomas-jefferson-35976004)
8. Edwin Black, *L'IBM e l'Olocausto*, Rizzoli, Milano, 2001

Il preprocessore

1. Jacobellis, 378 U.S. at 197.
   Citato da Paul Gewirtz, *On "I Know It When I See It"*
   digitalcommons.law.yale.edu/fss_papers/1706
2. Richard Dawkins, *Il Gene Egoista*, Mondadori, Milano, 2010
3. *"Se un maschio ha ereditato dal padre una coda lunga, è probabile che abbia ereditato dalla madre anche i geni che l'hanno indotta a scegliere la coda lunga del padre"*

Richard Dawkins, *L'Orologiaio Cieco*, Mondadori, Milano, 2016

La memoria

1. Nyogen Senzaki, Paul Reps, *101 Storie Zen*, Adelphi, Milano, 1973
2. Curiosa coincidenza: la dottoressa Kubler-Ross pubblicò la sua teoria sull'elaborazione del lutto nel 1969.
3. Come disse Karl Kraus: *"È ben nota la pretesa di avere un posto al sole. Meno noto è che il sole tramonta appena il posto è raggiunto".*
4. Un tipo di allenamento ideato dal dottor Kenneth Cooper per prevenire le malattie coronariche. Lo studio fu pubblicato, guarda caso, nel 1968.
5. L'equivalente di 160 Dollari, nel 2020.
6. *il piacere , che è del tutto egocentrico, sta nella percezione che ci si possa sottrarre ai propri simili, che ogni uomo, donna, bambino, paese o città non sia nient'altro che un punto di orientamento su una strada che porta all'isolamento completo – un isolamento dove non esiste più alcun obbligo sociale*
   Lady Jeune, baronessa St. Helier, sul piacere di guidare un'automobile nel 1904.
   Citato da Mathijs Deen, *Per antiche strade*, Iperborea, Milano, 2020
7. Il termine *Sakoku*, che vuol dire: *Paese chiuso*, indica il periodo della storia del Giappone che va dal 1641 al 1853, durante il quale il Giappone si chiuse quasi totalmente ai rapporti con l'estero.
8. Citato da Ananda K. Coomaraswami, *Tempo ed Eternità*, p. 22 Luni, Milano, 2003
9. @todo: Spiegare cosa sia la tipizzazione forte.
10. www.nobelprize.org/prizes/peace/1983/walesa/facts/

Le funzioni

1. isocpp.org/wiki/faq/templates#templates-defn-vs-decl
2. *Samyutta Nikaya*, Citato da Ananda K. Coomaraswami, *Tempo ed Eternità*, p. 45, Luni, Milano, 2003
3. *Una funzione può essere richiamata in qualsiasi punto di un programma e, per ripristinare lo stato della funzione chiamante, è necessario che il suo indirizzo di ritorno sia memorizzato in uno stack di tipo LIFO (Last In First Out), insieme a tutte le variabili locali. Ogni volta che una funzione termina il suo ciclo di istruzioni, i suoi dati vengono rimossi dallo stack, ma, nel caso della ricorsione, lo stack viene liberato solo al termine della ricorsione stessa e vengono memorizzati contemporaneamente tanti set di dati (indirizzo di ritorno della funzione e variabili locali) quante sono le chiamate ricorsive.*
   Claudio Munisso, Carlo Simonelli, *Dal C a Windows*, Logica, Roma, 1995

4. Eiji Yoshikawa, *Musashi*, Rizzoli, Milano, 1994
5. Secondo Angus Maddison, *The world economy: a millennial perspective*,

Development Centre of the Organisation for Economic Co-operation and Development, Paris, 2001, p.29, l'età media in Giappone fra il 1776 e il 1875 era di 32 anni; è ragionevole pensare che duecento anni prima fosse ancora più breve.

6. *Se tu ancora non hai ammirato la bellezza dell'Amico, alzati, non oziare, va alla ricerca dei Suoi segreti! Se ne sei ignaro devi cercare: fino a quando sa rai come un asino privo di briglie?*
   Farid al-Din 'Attar, *op.cit.*
7. Musashi Miyamoto, *Il libro dei cinque anelli (Go rin no sho)*,

Istruzioni condizionali

1. Sa^{c}di di Shirāz, poeta persiano vissuto fra il 1200 e il 1291. La poesia citata compare all'ingresso del Palazzo di Vetro dell'ONU.
2. Farid al-Din 'Attar, *op.cit.*
3. @todo: trovare riferimento bibliografico o quanto meno spiegare di cosa si stia parlando.
4. *"Nell'universo non esistono dèi , non esistono nazioni né denaro né diritti umani né leggi , e non esiste alcuna giustizia che non sia nell'immaginazione comune degli esseri umani"*
   Yuval N. Harari, *Sapiens. Da animali a dèi: Breve storia dell'umanità.* Bompiani, Milano, 2019
5. Il tasso di analfabetismo funzionale, mentre scrivo, è del 41%. I lettori sono sempre di meno: www.istat.it/it/archivio/213901
6. www.ancient-buddhist-texts.net
   @todo: verificare la traduzione del testo.
7. Farid al-Din 'Attar, *op.cit.*
8. Mumon, *Nansen Taglia il Gatto in Due*, da: *La Porta senza Porta*, Adelphi, Milano, 1987.

Istruzioni iterative

1. In effetti, sempre ne *La Metafisica*, disse anche che tutti gli uomini aspirano per natura alla conoscenza, quindi non è che sia tanto credibile, come fonte..
2. blackfridaydeathcount.com
3. Piera Scarabelli, Massimo Vinti - *op.cit.*
4. @todo: aggiungere definizione di connessione stateful
5. @todo: aggiungere definizione di connessione stateless
6. I ricercatori del progetto Openworm hanno mappato il cervello di un verme in un automa Lego. L'automa, se incontra un ostacolo sulla sua strada, si gira e torna indietro anche se il programma che lo controlla non contiene questa istruzione.

Le classi

1. Lévy-Bruhl, citato da Coomaraswamy, *Il Grande Brivido*, pag. 227

2. *However, although it is easier on the person who writes the class, it is harder on all the readers since it mixes what a class does (the external behavior) with how it does it (the implementation). Because of this mixture, you should define all your member functions outside the class body if your class is intended to be highly reused and your class's documentation is the header file itself*
   isocpp.org/wiki/faq/inline-functions#inline-member-fns.
3. Da C++17; non utilizzo questa sintassi nel testo perché preferisco una forma più portabile.
4. @todo: aggiungere citazione, credo sia *Evolution and the Theory of Games*

5. Andrea Pazienza (1956, 1988), illustratore, fumettaro, artista.
   www.andreapazienza.it
6. Agostino Straulino (1914, 2004), velista, Comandante dell'Amerigo Vespucci.
   www.treccani.it/enciclopedia/agostino-straulino
7. @todo: aggiungere definizione del termine
8. *Disse [Iblīs]: «Lasciami attendere [la tua punizione] sino al giorno in cui gli uomini saranno resuscitati». Rispose il Signore: «Ebbene, ti sia concesso di attendere sino a quel giorno». E disse ancora Iblīs: «Poiché Tu mi hai fatto errare, io mi apposterò sulla Tua Via Diritta e apparirò loro davanti, di dietro, a destra e a sinistra! E non certo molti di loro troverai che Ti saranno grati». E disse Iddio: «Fuori di qui, esci di qui spregiato e rejetto! Di quei che t'avranno seguito, di voi tutti empirò la Geenna!»* - Cor. VII 14-18
   Citato da Carlo Saccone, *Iblīs, il Satana del Terzo Testamento. Letture coraniche II*, Centro Essad Bey-CreateSpace IPP, Charleston 2016 (seconda ed.)
9. @todo: aggiungere riferimento ad Ananda, nota 35 a capitolo sull'Induismo.

L'ereditarietà

1. Giorgio de Santillana, Hertha von Dechend, *Il mulino di Amleto*, Adelphi, Milano, 1983
2. gianlucafusto.com
3. Perifrasi di Matteo 12:1-8.

Il polimorfismo

1. Il vero spirito tradizionale, quale si sia la forma da esso rivestita, è in fondo sempre e ovunque lo stesso; le forme diverse, specificamente adatte a queste o quelle condizioni mentali, a queste o quelle circostanze di tempo e di luogo, sono solo le espressioni di una unica e sola verità. René Guénon, *La crisi del mondo moderno.* Edizioni Mediterranee, Roma 2010.
2. Richard Dawkins, *L'Orologiaio Cieco*, Mondadori, Milano, 2016

3. Film del 1956, diretto da Camillo Mastrocinque, con Totò, Peppino De Filippo e Giacomo Furia.

Gli stream

1. @todo: Spiegare cosa sia una bitmask.
2. *Come nacque il gioco degli scacchi: due principi indiani, Gav e Talhend, si contendevano il regno, nonostante la loro madre li pregasse di condividerlo come bravi fratelli. I due non ci riuscirono e una loro lite si trasformò in una sanguinosa guerra, che venne infine vinta da Gav, e nella quale Talhend perse la vita. La madre, disperata, chiese al figlio superstite di raccontarle per filo e per segno come si fosse svolta la battaglia tra i due, e per spiegarle tutto quanto in modo dettagliato, Gav creò una scacchiera e fece intagliare nel legno le figurine di torri, cavalli e soldati, creando così le basi del gioco degli scacchi*
   Milla Fois, *Miti Persiani: Zoroastro e il Libro dei Re*
   Vedi anche il post: La Mossa del Cavallo.

Il debug

1. Benedetto Croce, *Breviario di estetica*, Laterza, Bari, 1928
2. @todo: Spiegare la differenza fra approccio *bottom-up* e *bottom-down*.
3. Mathijs Deen, op. cit.
4. *C'è una buona regola per gli storici che ho sempre tenuto presente quando lavoravo al mio* Galileo*: mai sottovalutare il potere della stoltezza e dell'insensatezza nelle vicende umane.*
   Giorgio de Santillana, Hertha von Dechend, op. cit.
5. @todo: Spiegare cosa sia il firmware e la differenza con il software.

Note

1. René Guénon, op. cit.
2. Personaggi del racconto di William Faulkner *L'Orso*.
3. Creatore del sistema operativo Linux.
4. Creatore del sistema operativo Minix, a cui Torvalds si ispirò per creare Linux.