

**ISTITUTO D'ISTRUZIONE SUPERIORE "G. ROMANI"  
CASALMAGGIORE**

**DISPENSE DI INFORMATICA**  
**Corso Informatica/Telecomunicazioni**  
**CLASSI 3<sup>^</sup> a.s. 2023-24**

**Teoria**

**Autori:**

**prof. Fabrizio Camuso**

**NOTA: alcuni contributi sono stati trovati su Internet ed in questo caso gli autori concedono pubblico utilizzo di solito a patto di citare le fonti come è stato ovviamente fatto.**

**Gli argomenti trattati congiuntamente con i contenuti scaricabili dai siti web dei professori, con possibilità di stampa per chi non avesse a disposizione un pc a casa) sono aderenti alle linee guida ministeriali.**

**In particolare molto altro materiale (esercizi aggiuntivi risolti, dispense, videolezioni) è disponibile sul sito web: [www.camuso.it](http://www.camuso.it) e da youtube (cercare l'utente *fcamuso o il suo canale 'Programmazione e dintorni'*)**





## Introduzione al corso

---

L'obiettivo di questo corso triennale è insegnarvi a progettare e realizzare soluzioni informatizzate per ogni esigenza dal semplice applicativo per uso personale al contribuire allo sviluppo di un sistema informativo di una grossa industria. Mentre una volta gli unici dispositivi elettronici coinvolti erano un singolo elaboratore elettronico, *stand-alone* direbbero gli anglosassoni, dotato solo di monitor, tastiera e (non sempre!) stampante oggi una soluzione informatizzata deve prevedere l'integrazione con un numero impressionante di altre periferiche: mouse, touch screen, telecamere, microfoni, altoparlanti, modem, router, gps, accelerometri, sensori di vario tipo (temperatura, luce, pressione ...), sistemi di memorizzazione esterni ecc. Non solo: deve essere in grado di scambiare informazioni e servizi con altri elaboratori elettronici collegati in rete locale (pensate ad un ufficio) o remota (pensate ad Internet).

Questo aumento di complessità ha due conseguenze immediate:

- la presenza nell'indirizzo di studi che avete scelto di molteplici discipline (elettronica e telecomunicazioni, tecnologie e progettazione di sistemi informatici, sistemi e reti, gestione progetto ed impresa, informatica) ciascuna focalizzata su aspetti specifici ma le cui **conoscenze e competenze devono essere usate insieme** per realizzare le sopra citate soluzioni informatiche
- la necessità di **sapersi organizzare in team** perché spesso lo sforzo ed il tempo richiesto per lo sviluppo di un applicativo software supera le possibilità di un singolo

*Iniziamo con il chiarire alcuni concetti...*

## Macchine ed elaboratori elettronici

---

Una **'macchina'** è un qualsiasi **qualsiasi dispositivo costruito dall'uomo che è in grado di eseguire delle operazioni almeno in parte automaticamente cioè idealmente senza l'intervento umano.**

Una macchina vera e propria è un qualche cosa di più rispetto ad un semplice strumento quale un martello o delle forbici (che potremmo al massimo considerare come forme minimali di macchine): è composta di solito da un buon numero di parti coordinate tra loro a svolgere un compito in modo almeno semi automatico.

Un esempio di macchina semi automatica è l'affettatrice di salumi a manovella; l'uomo fornisce la forza umana, deve regolare la macchina (distanziare la lama per ottenere fette sottili o spesse, eventualmente montare il giusto tipo di lama); ma il taglio in sé avviene poi in automatico e coinvolge diverse parti della macchina (la lama, il carrello su cui scorre avanti e indietro il salume, la parte che raccoglie e dispone le fette tagliate, i dispositivi di sicurezza ecc.).

Un esempio dove invece l'automatismo è pressoché totale è rappresentato da una lavastoviglie o un veicolo a guida autonoma.

Una macchina si dice elettrica quando la forza motrice è fornita appunto dall'elettricità (uso di **circuiti elettrici**). Il termine **circuito elettronico** implica invece qualche cosa in più: in questi circuiti **l'elettricità viene usata per rappresentare ed elaborare dati** dove per elaborazione si intende una trasformazione in altri dati; esempi di elaborazione sono: calcoli numerici, mettere in ordine alfabetico un elenco di nomi, ruotare sullo schermo una immagine o sostituire un suo colore.

## **Circuiti in logica cablata o programmata**

---

I primi circuiti elettronici svolgevano elaborazioni molto semplici e fisse: venivano assemblati per un compito e non potevano essere adattati per un altro. Si parla in questo caso di circuiti in **logica cablata o di soluzioni hardware**. Ad esempio le prime calcolatrici elettroniche automatiche (le antenate delle moderne calcolatrici tascabili) erano in grado di svolgere solo determinati calcoli e non altri.

**Un elaboratore elettronico** (quello che comunemente chiamiamo PC) invece non solo è una macchina automatica ed elettronica ma è anche **programmabile** cioè è in grado di eseguire blocchi di istruzioni, i **programmi**, che possono essere facilmente sostituiti a seconda dell'elaborazione da svolgere. I programmi sono forniti dall'esterno e memorizzati nella RAM (la memoria elettronica di lavoro) dell'elaboratore per il tempo necessario alla loro esecuzione e possono essere cambiati con altri in qualsiasi momento. I circuiti elettronici programmabili sono detti funzionare in **logica programmata**; e si parla di **soluzione software** (in contrapposizione alla logica cablata e relativa soluzione hardware descritta in precedenza).

**Questo significa che le soluzioni software sono migliori?** No: sono solo più flessibili; non sarebbe certo pratico ed economico avere un PC cablato solo per far funzionare un programma di videoscrittura ed un altro solo per usare un foglio elettronico ed un altro per ...

Ma quando è richiesta la massima velocità di elaborazione allora viene preferita una soluzione hardware e questa può anche essere incorporata in un sistema programmabile: ad esempio le moderne GPU (Graphic Processing Unit, cioè le schede video) fanno largo uso di circuiti in logica cablata per svolgere sempre le stesse operazioni grafiche anche in contemporanea su tanti dati (calcolo parallelo). Un player multimediale potrebbe sfruttare chip in grado di decodificare particolari formati audio video con prestazioni irraggiungibili da soluzioni software; apparecchiature medicali potrebbero sfruttare chip in grado di analizzare con il minor ritardo possibile dati clinici rilevati da sensori: si tratterebbe di chip in grado di fare solo quello ma alla massima velocità.

Vi starete probabilmente chiedendo cosa renda una soluzione software più lenta: le istruzioni (anche centinaia di milioni) che costituiscono un programma sono memorizzate nella RAM e devono essere lette una alla volta (e questo richiede tempo), interpretate (e questo richiede altro tempo) e poi inviati i segnali elettrici corrispondenti all'istruzione a tutte le unità di elaborazione coinvolte (altro tempo). In una soluzione cablata in un certo senso il programma è il circuito stesso! Una soluzione hardware non presenta quasi ritardi: forniti impulsi elettrici in ingresso al circuito quasi immediatamente vengono forniti quelli in uscita come risultato.

### Definizione completa di elaboratore informatico

Parlare di computer (calcolatore) è assai riduttivo. Più correttamente si deve parlare di **sistema di elaborazione elettronico di informazioni in formato digitale programmabile**.

*Sistema* in quanto costituito da molte parti ciascuna con la sua funzione specifica ma coordinate verso uno scopo comune, l'elaborazione delle informazioni.

*Elaborazione*: non solo calcoli (che certamente *sono* una forma di elaborazione) ma più in generale la trasformazione di dati in altri dati (ordinare un elenco di nomi, trasformare una immagine da colori a bianco e nero, conversione di un suono dal formato .wav al formato .mp3

*Elettronico*: i circuiti elettrici sono usati per rappresentare i dati e per svolgere la loro elaborazione.

*Digitale*: ogni tipo di informazione del mondo esterno è rappresentata nelle memorie del computer sotto forma di numeri secondo il sistema binario, cioè usando solo le cifre 0 e 1.

*Programmabile*: le funzioni svolte cambiano a seconda delle istruzioni caricate nel dispositivo.

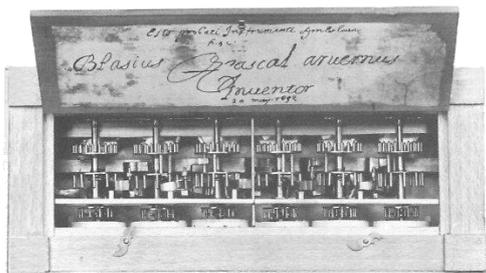
## Un po' di storia – prima dell'era moderna

La strada per arrivare a dispositivi che rispettino la definizione completa di sistema di elaborazione è costellata di ingegnosi passaggi intermedi sia a livello di hardware che teorici. Inizialmente e per molti secoli gli antenati degli attuali elaboratori si sono contraddistinti per le seguenti caratteristiche:

- pur stupendo per la loro ingegnosità e complessità costruttiva erano completamente **meccanici** e la forza che li faceva funzionare era umana o comunque non elettrica
- **non erano programmabili** ma costruiti in modo da poter svolgere un solo tipo di elaborazione e cioè calcoli matematici
- non rappresentavano le informazioni secondo il sistema binario

Alcuni esempi degni di nota di **dispositivi di calcolo meccanici**

### La Pascalina (dal genio di **Pascal - 1642**)



Poteva svolgere solo somme e sottrazioni

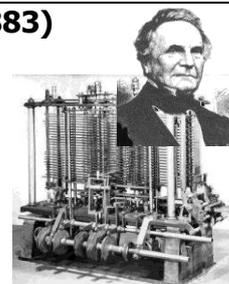
Potete seguire una fantastica animazione in 3D della Pascalina in azione al seguente indirizzo:

<https://www.youtube.com/watch?v=3h71HAJWnVU>



### La macchina di **Babbage (1833)**

Pur in forma meccanica aveva tutte le parti di un moderno calcolatore (unità di elaborazione, memoria, input/output) e per questo motivo Babbage è considerato il 'padre' del computer; non usava però la numerazione binaria ma la tradizionale decimale.



Ai seguenti indirizzi è possibile trovare delle fantastiche animazioni 3D della macchina:

[https://www.youtube.com/watch?v=vdra5Ms\\_9s](https://www.youtube.com/watch?v=vdra5Ms_9s)

<https://www.youtube.com/watch?v=sklp9WutBhs>



Il primo elaboratore meccanico con logica binaria completo e funzionante lo dobbiamo a Zuse (lo Z1, 1938)



<https://www.youtube.com/watch?v=duOBnyUdT2M>

**Usava l'elettricità solo come forza motrice** (anche se dotato di una manovella, visibile sulla sinistra, questa richiedeva troppa forza).

**Turing completa.**

## Avanzamenti teorici

I primi studi sul sistema di numerazione binario (rappresentazione dei numeri, operazioni binarie, conversione binario/decimale) sono dovuti a **Leibniz** (1679) ma vennero considerati bizzarrie!

Le sue idee saranno però riprese da **George Boole** (1815-1864): a questo matematico si deve **l'algebra booleana** (anch'essa considerata una bizzarria dagli scienziati del tempo) che rappresenta la base teorica del funzionamento dei circuiti elettronici: questa disciplina studia le proprietà degli insiemi formati da soli due elementi, vero e falso (assimilabili all'uno e allo zero del sistema binario di Leibniz!) definendo un gruppo di operatori (and, or e not assimilabili alle operazioni binarie di Leibniz). Boole dimostrerà come usando l'algebra Booleana sia possibile trattare un ragionamento logico come se fosse un calcolo da svolgere in modo meccanico con carta e penna.

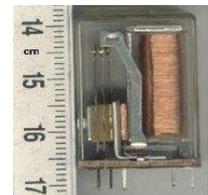
Oggi ci rendiamo conto che senza gli studi di Leibniz e di Boole l'informatica semplicemente non esisterebbe: i loro sono infatti esattamente i concetti alla base dei circuiti elettronici (invece della carta e della penna di Boole) su base binaria.

## L'elettricità non solo solo come forza motrice ...



Nel 1937 il matematico **Claude Shannon** dimostrò infatti che le operazioni booleane invece di essere svolte lentamente e con errori da un umano potevano essere svolte da circuiti elettrici in cui lo zero e l'uno erano rappresentati dallo stato aperto/chiuso di un *relè* con possibilità di commutare cioè di passare da uno all'altro.

In un relè un elettromagnete può aprire/chiudere un circuito per far passare (1) o non far passare (0) elettricità. Una commutazione da '0' a '1' avviene in un centesimo di secondo.



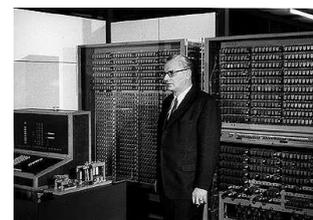
Nel 1948 Shannon pubblicò inoltre uno studio che costituisce le fondamenta della moderna Teoria dell'Informazione (argomento che approfondirete nel corso di Tecnologie).

## Il primo elaboratore digitale (binario) programmabile elettromeccanico

Zuse aveva già realizzato (1938) lo Z1 **il primo elaboratore meccanico programmabile e binario** con tutte le capacità di elaborazione di un moderno elaboratore (Turing completo).

Nel 1941 si ispirò ed ampliò il lavoro di Shannon realizzando lo Z3 sostituendo alle parti meccaniche del suo precedente Z1 con dei relè.

Lo Z3 è il primo vero computer digitale, programmabile, basato sul sistema binario e completamente automatico (anche se non ancora elettronico in quanto i relè hanno una parte meccanica).

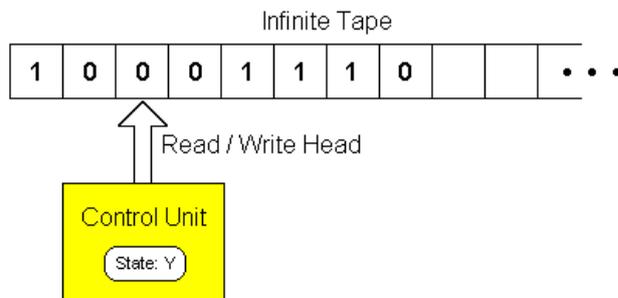


**MA** in che modo possiamo affermare che un dispositivo sia in grado di svolgere qualunque tipo di elaborazione affermando che è equivalente, solo più lento, a un moderno elaboratore? Per questo dobbiamo scomodare il padre dell'informatica teorica, **Alan Turing**.

Nel 1937 il geniale matematico **Alan Turing** pubblicò un trattato in cui descriveva una macchina astratta (**MdT**, la macchina di Turing appunto) che fissò i limiti di ciò che poteva essere elaborato da un computer. La genialità è dovuta alla **rigorosa dimostrazione matematica** che la MdT può svolgere qualunque elaborazione su dati binari. Turing è considerato dagli informatici teorici il fondatore della loro scienza.



Per astratta si intende che non è necessario costruirne una e che il suo funzionamento può essere descritto anche con carta e penna ottenendo gli stessi risultati di una macchina fisica. Il funzionamento di questa macchina è semplicissimo ma è in grado di svolgere *qualunque* tipo di elaborazione. Essendo pochissime e semplicissime le operazioni che può eseguire (vedi sotto) per ottenere lo stesso risultato anche di una sola istruzione di un linguaggio come il C++ serviranno molte operazioni della MdT. Nonostante questa semplicità Turing ha *dimostrato matematicamente* (in questo sta la sua grandezza) che nessun computer presente o futuro potrà essere più capace della sua MdT.



Una MdT è costituita da un nastro, ipoteticamente infinito, suddiviso in cellette ciascuna delle quali rappresenta un bit (è la memoria della MdT, la sua "RAM") e può contenere infatti 1, 0 o uno spazio (essere vuota insomma).

La "CPU" è costituita da una testina di lettura/scrittura sul nastro che in base allo stato in cui si trova (stato 0, 1, 2, 3 ecc.) e al simbolo che legge dal nastro può o

scrivere 1 o 0 o cancellare il contenuto della cella del nastro oppure spostarsi di una posizione a destra o a sinistra. La MdT inizierà sempre dalla prima cella a sinistra del nastro.

Le MdT più semplici corrispondono a un tipo di elaborazione fisso (MdT in grado di moltiplicare due numeri e solo questo ad esempio); ma una MdT universale è pensata per svolgere qualunque tipo di elaborazione.

Adesso immaginate di far trovare pronta sul nastro la sequenza di 1 e 0 che rappresentano i dati da elaborare; sembra incredibile ma QUALUNQUE elaborazione, le stesse portate a termine dal più complesso microprocessore oggi esistente e che mai esisterà, potranno essere espresse come una (lunghissima) sequenza delle micro-operazioni di una MdT universale.

In che modo può essere sfruttato questo strumento teorico?

- Quando viene progettato un nuovo tipo di dispositivo di elaborazione per avere la certezza che sia in grado di calcolare tutto il possibile è sufficiente dimostrare che è in grado di simulare il funzionamento di una MdT universale e il dispositivo sarebbe definito allora *Turing completo*; usare come confronto un sistema reale complicherebbe MOLTO la dimostrazione
- Forse vi sorprenderà sapere che esistono dei problemi che non possono essere risolti con il computer. Ebbene anche il dimostrare che un problema non è 'computabile' (calcolabile) è un risultato importante: se si dimostra che non può essere concepita una

macchina di Turing che computa il risultato allora ciò equivale a dimostrare che nessun computer potrà essere mai usato per risolvere quel problema (quanto meno risolverlo in modo completo: ci dovremo accontentare di una soluzione approssimata). Anche in questo caso una dimostrazione che usasse un normale PC come riferimento sarebbe MOLTO più difficile. Detto questo vi tranquillizzo dicendovi che la stragrande maggioranza dei problemi che ci interessano è computabile.

#### NOTE.

E' stato dimostrato che tutti gli elaboratori di Zuse sono turing-completi: cioè non potrà mai essere costruito un elaboratore in grado di risolvere problemi che non possano essere risolti anche con uno Z1 o uno Z3: l'unica differenza sarà nelle prestazioni!

Pare che Neumann nel progettare la struttura degli elaboratori ancora oggi in uso abbia tenuto come riferimento la MdT universale

### Un po' di storia – era moderna

---

Quella vista fino ad ora può essere considerata la preistoria dei moderni calcolatori. Siamo pronti per il passaggio dall'elettrico (esistono ancora parti meccaniche di elaborazione) all'elettronico (assenza di parti meccaniche negli organi di elaborazione). Vengono individuate quattro generazioni. Le parole d'ordine sono: miniaturizzazione sempre più spinta, sempre più maggiore velocità di elaborazione, costi decrescenti.

Ogni passaggio di generazione è infatti caratterizzato da un salto tecnologico nella rappresentazione degli stati 1/0 con tecnologie che consentono velocità di commutazione tra gli stati sempre più elevate, con dispositivi sempre più integrati e miniaturizzati e sempre più economici da realizzare.

#### *Prima generazione 1945 - 1955 (valvole termoioniche)*

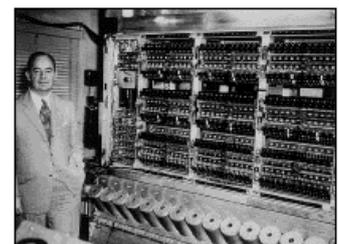


L'innovazione che caratterizza la prima generazione è l'utilizzo delle **valvole termoioniche** al posto dei relè per rappresentare gli stati 0 e 1. I passaggi di stato 0/1 risultavano molto più veloci che con i relè.

Negli USA il **primo calcolatore completamente elettronico** che usa queste valvole venne realizzato dai fisici **Atanasoff e Berry** nel 1942: fu chiamato ABC (**A**tanassoff **B**erry **C**omputer). L'ABC non era però Turing completo: questo primato spetta a ENIAC (1946).



Nel 1952, l'ungherese **Von Neumann** progettò e realizzò l'EDVAC la cui architettura interna (architettura di Von Neumann) divenne la pietra miliare per la costruzione dell'hardware dei calcolatori elettronici: fu la prima che **prevedeva il programma registrato internamente** (architettura di Von Neumann). L'intuizione di Von Neumann fu quella di **esprimere le istruzioni per il calcolatore nella stessa forma dei dati**, cioè attraverso un codice numerico opportuno, e di immagazzinarle in memoria prima dell'avvio dell'elaborazione.



### ***La seconda generazione 1955-1964 (transistor)***

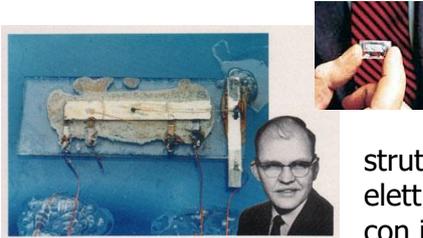


A partire dalla seconda metà degli anni '50, le valvole termoioniche cominciarono ad essere sostituite con i **transistor**, (Brattain, Shockley, Bardeen, premiati per questo con il Nobel nel 1956). I transistor presentavano il vantaggio di essere più economici, molto più piccoli, decisamente più affidabili (molte meno avarie rispetto alle valvole) e, infine, di consentire un aumento di 10 volte della velocità di elaborazione. Un transistor funziona come un interruttore e con un impulso elettrico di controllo può essere fatto transitare in uno stato che possiamo interpretare come un 1 oppure uno 0. Il transistor può allora essere sfruttato per creare celle di memoria o porte logiche come dire tutto ciò di cui si ha bisogno per costruire dispositivi di elaborazione elettronica.

Nascono in questi anni i primi sistemi operativi ed i primi linguaggi ad alto livello (COBOL, Fortran). Il computer diventò molto più accessibile ad una vasta gamma di attività e si diffuse in decine di migliaia di esemplari in tutto il mondo.

### ***La terza generazione 1964 -1970 (circuito integrato)***

Da un punto di vista *hardware*, la "terza generazione" si caratterizzò per l'impiego dei **circuiti integrati o chip**, per lo sviluppo di memorie a dischi magnetici di grande capacità e per l'utilizzo dei primi collegamenti a distanza fra gli elaboratori.



Il circuito integrato, inventato nel 1958 da J. St. Clair Kilby, della Texas Instruments, è costituito da una piastrina di silicio di pochi millimetri quadrati su cui vengono realizzate strutture in grado di funzionare come transistor e altri componenti elettronici come resistenze e condensatori. La grossa differenza con i circuiti tradizionali è che non ci sono parti assemblate, saldate tra loro: il circuito viene 'stampato' nel silicio.

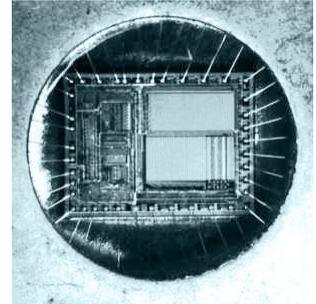
Con il passare del tempo, la quantità di componenti contenuti in un chip aumenterà progressivamente, tanto che il loro grado di integrazione diventerà il criterio in base al quale distinguerli in classi: dopo i primi esemplari degli anni '60 chiamati SSI (Small Scale Integration), con qualche decina di componenti, e MSI (Medium Scale Integration) fino a 500 componenti, si passerà negli anni '70 agli LSI (Large Scale Integration) con alcune decine di migliaia di componenti. Seguiranno anche la VLSI (Very ... centinaia di migliaia) e la ULSI (Ultra ... centinaia di milioni).

L'adozione dei circuiti integrati negli elaboratori della "terza generazione" consentì una drastica riduzione di costo, peso ed ingombro, ed una velocità di calcolo dell'ordine dei nanosecondi (miliardesimi di secondo).

## ***La quarta generazione (1971 – oggi) e la nascita del personal computer***



Nel 1971 ebbe inizio una seconda rivoluzione industriale grazie all'invenzione, da parte di Federico Faggin (un italiano!), M. E. Hoff e S. Mazer, del **microprocessore**, un "supercircuito" integrato, impiantato su una piastrina di 4,3 millimetri, contenente ben 2.250 transistor che costituiscono **tutti i componenti di una unità centrale di elaborazione su un solo chip**.



Questo *microprocessore*, l'Intel 4004, con parallelismo a 4 bit e una capacità di calcolo di 60 mila operazioni al secondo, segnò l'inizio della quarta generazione di elaboratori che è ancora quella in corso (una quinta, profetizzata decenni fa come imminente dai giapponesi ed imperniata sull'uso diffuso di computer intelligenti si è rivelata molto più lontana nel tempo e gli obiettivi che si poneva assai più difficili del previsto da raggiungere).

Nel 1976 due giovani intraprendenti californiani, S. Wozniak e S. Jobs, progettarono e costruirono in casa, con il *microprocessore* 6502 della MOS Technology, il **primo personal computer**, l'APPLE I (dotato di ambiente di sviluppo per linguaggio BASIC), che ottenne uno strepitoso successo.

Nel 1981, l'IBM (ricordate il signor Hollerith?) entrò nel mercato dei personal computer con il modello PC-IBM equipaggiato con *microprocessore* Intel 8088 a parallelismo a 8 bit, 64 Kb di RAM ed un floppy disk drive da 5 ¼ pollici.

Nel 1985 l'**Intel** presentò il *microprocessore* 386 a 32 bit a cui seguì il 486 ed il 586 (Pentium). **AMD** introduce a questo punto sul mercato dei processori compatibili a prezzi vantaggiosi e da allora la sfida per le prestazioni ha visto più volte scavalcarsi tra loro le due ditte.

Oggi stiamo vivendo l'era del multiprocessore (*multicore*): chip che ospitano diversi interi microprocessori che lavorano in parallelo nello spazio prima occupato da uno solo; sono commercialmente disponibili soluzioni *otto core* (8 nuclei di elaborazione) e più a prezzi assolutamente accessibili. Come si suol dire: il resto è storia dei nostri giorni...

### **I System On Chip (SOC)**

Un SOC è un unico circuito integrato a basso consumo che realizza le funzioni di tutti i componenti che costituiscono un classico sistema di elaborazione (microprocessore, RAM, Input/Output, Storage, Networking).

#### Esempi

- Qualcomm Snapdragon (smartphones)
- Tegra X1 (Nintendo Switch)
- Raspberry PY

## Problemi, soluzioni e programmi; dati e informazioni

E' importante sottolineare che **una macchina non risolve un problema**: essa si limita ad eseguire la giusta sequenza di **istruzioni** determinata però dal vero **risolutore che è sempre un umano**.

E' quindi il **programmatore** che fornisce la sequenza di istruzioni (la soluzione al problema) che verrà poi eseguita in automatico dall'elaboratore elettronico. In uno spot pubblicitario (pneumatici) si affermava: *la potenza è nulla senza controllo*. Potreste avere il computer più potente del mondo ma senza **software** non potreste servircene; avreste solo un **hardware** (la parte materiale: processore, scheda video, ram, hard disk ecc.) incapace di portare a termine un qualsiasi compito. Mancherebbero le **istruzioni** (*instruction*). Ecco perché tutti i nuovi elaboratori elettronici sono di solito già equipaggiati al momento dell'acquisto almeno con un primo sofisticato *software*: il sistema operativo (*Windows, Linux, MAC OS, Android* ecc.); approfondirete l'argomento sistemi operativi nelle discipline *Tecnologie* e *Sistemi e Reti*.

Un blocco di istruzioni espresse in una forma comprensibile per un elaboratore elettronico viene chiamato **programma** (*program*). Se per un certo dispositivo è possibile cambiare all'occorrenza il blocco di istruzioni da eseguire esso si dice **programmabile** (*programmable*; *corrisponde al concetto di logica programmabile e soluzione software cui si è accennato in precedenza*). Ovviamente qualunque PC o *smartphone* o *console* per videogame è programmabile.

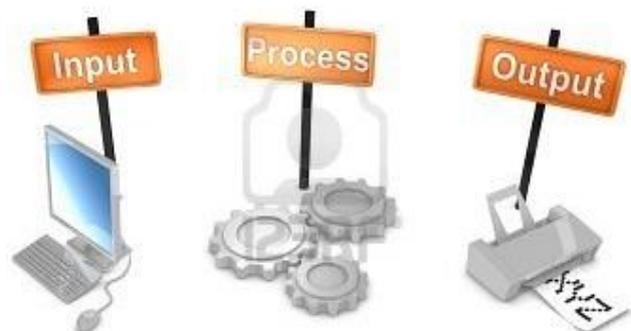
### Dati, informazioni e codifica digitale

Intuitivamente possiamo definire un'informazione come un dato (un numero, un simbolo ecc.) a cui sappiamo attribuire un *significato*.

Uno stesso numero (il dato) può rappresentare cose diverse: un peso, un'età, una distanza, il codice di un colore o la componente di un suono; ed una stessa serie di simboli (il dato) può significare cose diverse a seconda della lingua; ad esempio *case* in italiano è il plurale di casa mentre in inglese può significare contenitore o caso. Se non sappiamo il suo significato, il dato è inutile.

L'informazione può presentarsi in molte forme: numeri (*number*), testo (*text*), immagini (*image*), suoni (*sound*), filmati/animazioni (*video*). Ma è estremamente importante capire che qualunque sia il tipo di informazione questa viene codificata in formato binario, cioè digitalizzata, e sostanzialmente trattata come un numero (approfondirete questi aspetti nella materia *Tecnologie*).

**Ogni programma essenzialmente accetta in ingresso (input) dei dati o informazioni in forma digitale, procede ad una loro elaborazione (processing) e produce in uscita (output) altri dati e informazioni.**



## Linguaggi di programmazione

---

Ma come vengono fornite le istruzioni ad un elaboratore? Sono stati ideati degli appositi **linguaggi detti di programmazione** (*programming languages*). Questi linguaggi, pur artificiali, hanno le stesse caratteristiche di base di quelli naturali (come l'italiano o l'inglese):

- un **alfabeto** di simboli (symbols) più o meno corrispondente al nostro alfabeto sia minuscolo (lower case) che maiuscolo (upper case), le cifre da 0 a 9 (digit), la punteggiatura ed altri simboli speciali (come # o @)
- combinando i simboli si ottengono le **parole** (keyword) utilizzabili nelle istruzioni; ad esempio *print* la keyword che in diversi linguaggi comanda la 'stampa' sul video di un messaggio
- **regole** (sintassi, *syntax rules*) che fissano in modo rigoroso come le singole parole possono essere combinate a formare istruzioni che hanno senso per l'elaboratore elettronico; esattamente come le regole sintattiche della lingua italiana fissa le regole per comporre frasi ben formate in italiano
- **semantica**: è il 'significato' di ogni istruzione cioè gli effetti della sua esecuzione

### *Scelta del linguaggio e codifica del programma*

Dopo aver ideato il procedimento risolutivo di un problema il programmatore deve provvedere alla **codifica (coding)** cioè alla scrittura delle istruzioni scegliendo il linguaggio di programmazione che ritiene essere il più adatto.

Infatti alcuni linguaggi sono particolarmente efficaci per problemi scientifici mentre altri per problemi commerciali, altri per gestire oggetti multimediali ed altri ancora vanno abbastanza bene per qualsiasi compito.

In ogni caso la potenza espressiva dei linguaggi si equivale: se con un linguaggio si può scrivere un programma che risolve un problema allora con un qualunque altro linguaggio si potrà scrivere un programma che risolve lo stesso problema; quella che potrebbe cambiare è la difficoltà con cui lo si può fare proprio come capita nella vita di tutti i giorni scegliendo uno strumento più o meno adatto al compito da svolgere: si può andare a 1000 km di distanza anche in bicicletta o costruire piccoli castelli di sabbia usando una ruspa ma in nessuno dei due casi probabilmente si sta usando lo strumento giusto.

Quando un linguaggio di programmazione è specialistico in quanto favorisce uno specifico campo applicativo (come il Fortran, molto adatto per l'ambito scientifico) viene definito **special purpose** (da *purpose* = fine, obiettivo); quando invece un linguaggio si adatta a qualsiasi scopo senza eccellere in nessun ambito (come il C++) viene definito **general purpose**. Oggi vengono usati per lo più linguaggi **general purpose** probabilmente perché ci vogliono alcuni anni per diventare esperti nell'uso di un linguaggio e conoscerne uno che va abbastanza bene per ogni situazione è certamente un vantaggio non da poco. Esistono centinaia di linguaggi di programmazione! Cerca su Internet.

## Evoluzione dei linguaggi di programmazione



Nella sua forma più comprensibile per la CPU un programma è una sequenza di valori 0 ed 1. Un programma in questa forma usa quello che è chiamato **linguaggio macchina (machine language)**.

Si parla anche di **linguaggio basso livello (low level language)** perché è vicino alla macchina e lontano dal nostro linguaggio naturale.

La realizzazione e la manutenzione di un programma in linguaggio macchina è una sfida notevole:

- scrivere e modificare programmi in linguaggio macchina è molto difficile
- dato che le istruzioni sono così a basso livello, vicino all'*hardware*, è necessario conoscere alla perfezione anche quest'ultimo
- se cambia il *microprocessore* (la CPU), e con esso l'insieme delle istruzioni accettate, l'intero programma deve essere riscritto;

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0001000 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Nonostante queste difficoltà i primi abbozzi di sistemi operativi e le prime applicazioni sono stati scritti proprio così ... direttamente linguaggio macchina!

### Linguaggio Macchina (machine code)

#### SVANTAGGI

- Bit / Byte oriented, bassa astrazione
- Non strutturato
- Legato alla CPU (instruction set diversi)
- Codice legato all'HW
- Costringe a conoscere bene l'HW
- Poco o per nulla portabile

**E' DIFFICILE PROGRAMMARE**



#### VANTAGGI

- Controllo assoluto sull'HW
- Ottimizzazione in spazio e tempo
- Nessun bisogno di traduzione

Un primo miglioramento è stato introdotto con l'uso di **codici mnemonici** al posto delle sequenze di 1 e di 0. Il concetto è assai intuitivo: è molto più semplice ricordare qualche cosa del tipo 'SUM 15 21' che non '100100101011', per indicare che vanno sommati 15 e 21. Il programmatore può ora scrivere il programma usando un linguaggio un po' più lontano da quello macchina e più vicino al suo modo di esprimersi. Questo linguaggio è chiamato **assembly**.

Ora però prima di poter sottoporre il programma al computer è necessario un processo di traduzione: i codici mnemonici ed i valori espressi in decimale non sono comprensibili dalla CPU ! Insomma il 'SUM 15 21' deve essere ritrasformato in '100100101011', ricordate? Un programma apposito chiamato **assemblatore (assembler)**, questo sì scritto in linguaggio macchina poiché alla fine della catena non può che esserci un qualche cosa che scritto in linguaggio macchina..., traduce ogni codice mnemonico e valore decimale nella corrispondente sequenza di 1 e di 0. I linguaggi *assembly* devono comunque essere considerati, come il linguaggio macchina, di basso livello.

	BEGIN		;inizio programma
START:	IN	1	;lettura dato da unità 1
	JPZ	FINE	;controllo se il dato letto è 0
	ADD	RIS	;somma il dato a RIS
	STA	RIS	;memorizza la somma in RIS
	JMP	START	;torna a leggere un nuovo dato
FINE:	LDA	RIS	;il ciclo di lettura e somma è finito
	OUT	2	;scrittura del risultato sull'unità 2
	HLT		;fine elaborazione
RIS:	WRD	0	;voce per mantenere somma parziale
	END	START	;fine programma

Qui a lato, un esempio di programma scritto in un linguaggio *assembly* (non cercate di capirlo, è troppo presto!).

I linguaggi *assembly* sono comunque ancora troppo scomodi e soffrono, in fondo, ancora di tutti i problemi elencati per il linguaggio macchina: hanno reso solo un poco più lieve la vita al programmatore ma sono ancora molto legati alla CPU usata e difficili da usare.

Il passaggio successivo è stato lo sviluppo di linguaggi ancora più lontani dal linguaggio macchina e, di conseguenza, più vicini al nostro modo di esprimerci. Sono nati i **linguaggi ad alto livello (high level language)**. Questi sono caratterizzati da istruzioni molto espressive e allo stesso tempo facili da ricordare, ciascuna delle quali può anche corrispondere a centinaia di istruzioni in linguaggio macchina (diventa quindi più arduo il compito del programma traduttore!).

```
int totale=0, dato=0;

do
{
    cin >> dato;
    totale = totale + dato;
}
while (dato!=0);
```

```
cout << totale;
```

Ecco qui a lato la versione dello stesso algoritmo scritto in assembly visto prima codificata con il linguaggio di programmazione C++.

Il programmatore dichiara i simboli (totale, dato) che vorrà utilizzare specificando anche che sono dei numeri interi (int) ed assegnando loro un valore iniziale (zero).

Poi si ripetono (do = fai, esegui) due istruzioni finchè (while) il dato introdotto è diverso (!=) da zero; le due istruzioni sono la lettura (cin, da 'in' serire) del dato e la sua aggiunta al totale. Terminata la ripetizione si provvede alla scrittura del

risultato calcolato (cout, da 'out' cioè uscita).

Programmando con un linguaggio così il programmatore non deve conoscere i meccanismi *hardware* della stampante o della CPU: Cambia la stampante ? I comandi rimangono sempre quelli! Cambia la CPU ? Cambia addirittura il tipo di computer? Non devo modificare o riscrivere tutti i programmi: il C++ rimane C++ !!

Come ho già accennato esistono centinaia di linguaggi: C, C++, C#, Visual Basic, Java, Fortran, Cobol, Pascal, PHP, Perl, Python giusto per citare i più diffusi ... Per ciascuno di questi è necessario un **traduttore** (translator). Approfondiremo ora questo aspetto.

## **Traduttori: interpreti e compilatori**

---

Esistono due tipi di traduttori per linguaggi a medio/alto livello, interpreti e compilatori:

### 1. **Interpreti (interpreter)**

*L'interprete è un programma che deve sempre risiedere in memoria insieme alle istruzioni che deve eseguire occupando quindi una certa quantità di RAM per sé stesso (cosa non necessaria, come vedremo, con i compilatori).*

Quando si comanda la partenza del programma l'interprete considera le istruzioni dal codice sorgente una alla volta e non opera una vera traduzione permanente (non si preoccupa di generare un eseguibile in linguaggio macchina) ma si limita a riconoscere ciò che ogni istruzione richiede ed a compiere l'azione corrispondente. Nota: per rendere più efficiente l'esecuzione l'interprete potrebbe prima trasformare il sorgente in una forma intermedia (byte code) tra il sorgente e il linguaggio macchina; questa trasformazione può avvenire in tempi molto minori rispetto ad una vera traduzione in linguaggio macchina (compilatore) e non snatura le caratteristiche di un interprete.

Che il programma consti di 10 righe o di 10 milioni otteniamo quindi la partenza immediata dell'esecuzione essendo del tutto assente la parte della traduzione in linguaggio macchina (potenzialmente molto lunga) dei compilatori; potremmo anche far partire il programma da un punto qualsiasi senza dover aspettare la traduzione della parte che precede perché semplicemente la parte che precede viene letteralmente ignorata. Questo è molto comodo mentre si sta sviluppando un programma: significa aggiungere istruzioni e provarle subito senza attese che possono diventare davvero lunghe (vedi compilatore).

Esempi di interpreti molto usati:

- ogni browser web incorpora un interprete per il linguaggio Javascript
- ogni server web ospita alcuni interpreti di linguaggi; degno di nota per noi quello per il linguaggio PHP
- word, excel, powerpoint (e i loro corrispondenti open source di Open/Libre Office) ospitano un interprete per il linguaggio Visual Basic rendendo questi applicativi programmabili (permettono all'utente di definire delle proprie procedure incorporandole nel documento e di renderle disponibili con i tradizionali controlli visuali come bottoni, drop down list ecc.)
- Autocad ospita un interprete per il linguaggio Lisp consentendo di automatizzare procedure di disegno
- Tutti i principali sistemi operativi rendono disponibile un ambiente di interpretazione di comandi di sistema anche in forma di script con le tipiche strutture di programmazione per automatizzare la gestione dell'elaboratore

*Un programma interpretato è più lento nel completare i suoi compiti rispetto ad un eseguibile creato con un compilatore*

Se un'istruzione deve essere eseguita più volte (come nei cicli) verrà reinterpretrata altrettante volte perché l'interprete non tiene traccia del passato e questo comporta una perdita di tempo potenzialmente considerevole.

*Inerterprete sempre in RAM*

Un altro svantaggio dell'interprete è che DEVE essere sempre presente in memoria insieme al programma da interpretare. Se questo non comporta grossi inconvenienti su sistemi con svariati GigaByte di RAM lo stesso potrebbe non essere vero su microsistemi ove la RAM potrebbe misurarsi in pochi MegaByte anche se il progresso tecnologico sta rendendo trascurabile questo fattore riuscendo a dotare di incredibili quantità di memoria dispositivi anche molto piccoli.

*Poca tutela della proprietà intellettuale*

Una caratteristica potenzialmente problematica degli interpreti è che sul PC ove funziona l'interprete deve essere presente il sorgente del programma a completa disposizione di chiunque. Ma il sorgente in chiaro è il bene più prezioso per un programmatore. Immaginate di aver trovato un algoritmo per comprimere i dati molto di più e con molto meno tempo di programmi come WinZip, WinRAR o 7Zip. Quell'algoritmo può farvi diventare letteralmente ricchi e scegliere di farlo usare in forma interpretata sarebbe un puro suicidio commerciale: chiunque potrebbe accedere ai segreti del vostro codice!

## 2 **Compilatori (compiler)**

*Attesa prima di iniziare l'esecuzione*

Questi sono dei veri traduttori: il compilatore prima che inizi l'esecuzione traduce in linguaggio macchina \*tutte\* le istruzioni, l'intero programma. Se il codice è composto da milioni o decine di milioni di righe prima di poter far partire l'esecuzione potrebbero passare delle ORE. Lo sa bene chi si diletta a personalizzare una distribuzione di Linux! Non tanti anni fa con PC basati su Intel 386 potevano occorrere GIORNI. Anche senza arrivare a questi casi limite con programmi lunghi anche il classico errore di dimenticanza di un punto e virgola o di una parentesi 'si paga' caro in termini di tempo sprecato ...

Non fatevi ingannare dai pochi secondi che sperimentate in laboratorio o sul vostro portatile usando Code::Blocks o simili: è solo dovuto al fatto che si tratta di programmi 'giocattolo'. In ogni caso anche per programmi che potremmo definire già interessanti ed utili i tempi di attesa sono di solito limitati a poche decine di secondi grazie alla potenza degli attuali elaboratori.

*Il compilatore non è necessario per l'esecuzione*

Terminata la traduzione il compilatore non è più necessario per l'esecuzione che può essere attuata direttamente dalla CPU essendo la traduzione puro linguaggio macchina.

*Il massimo della velocità (di esecuzione)*

Un grosso vantaggio del compilatore è che l'esecuzione potrà avvenire alla massima velocità consentita dal microprocessore (è infatti assente la continua interpretazione anche di istruzioni già eseguite in precedenza come accade con gli interpreti). Potremmo sintetizzare in: partenza più lenta (per i tempi di traduzione) ma completamento delle elaborazioni svolte dal programma nettamente più veloce (anche di un fattore pari a cento). Per cui con algoritmi che prevedono pesanti cicli di ripetizione delle stesse istruzioni o che devono essi stessi essere ripetuti molte volte è sicuramente da preferire la compilazione.

*Proprietà intellettuale al sicuro*

Per l'utente finale (che non deve ovviamente attendere i tempi di compilazione ma riceve solo l'eseguibile finale) è sempre meglio poter disporre di un programma compilato. Inoltre un programma in forma compilata (=linguaggio macchina) è estremamente difficile da studiare; quasi impossibile risalire alla logica di un algoritmo complesso così da poterla copiare e sfruttare in altri programmi.

### 3. Interpretazione con IL e JIT (compilazione Just In Time)

Linguaggi come Java hanno fin dall'inizio adottato una tecnica per così dire 'ibrida'. Lo sviluppatore usa un compilatore che invece di tradurre in linguaggio macchina puro produce un codice intermedio (**IL Intermediate Language**) molto più compatto del codice sorgente e che ha già superato tutti i controlli e molte delle fasi tipiche della compilazione tradizionale. Questa traduzione è comunque nettamente più veloce di una completa in linguaggi macchina. Potremmo pensare a questo codice intermedio come ad un cibo precotto, non ancora pronto per la consumazione ma quasi. Sull'elaboratore che eseguirà l'applicativo deve essere presente un'interprete (la JRM, Java Run Time Machine oppure ) che interpreterà il codice IL molto velocemente (il codice , ricordate, è 'precotto') guadagnando in velocità rispetto ad un interprete tradizionale e mantenendo molti dei vantaggi. Un po' come se partendo da istruzioni molto descrittive producessimo una sintesi compatta ma completa: seguendo la sintesi saremo più veloci nell'operare.

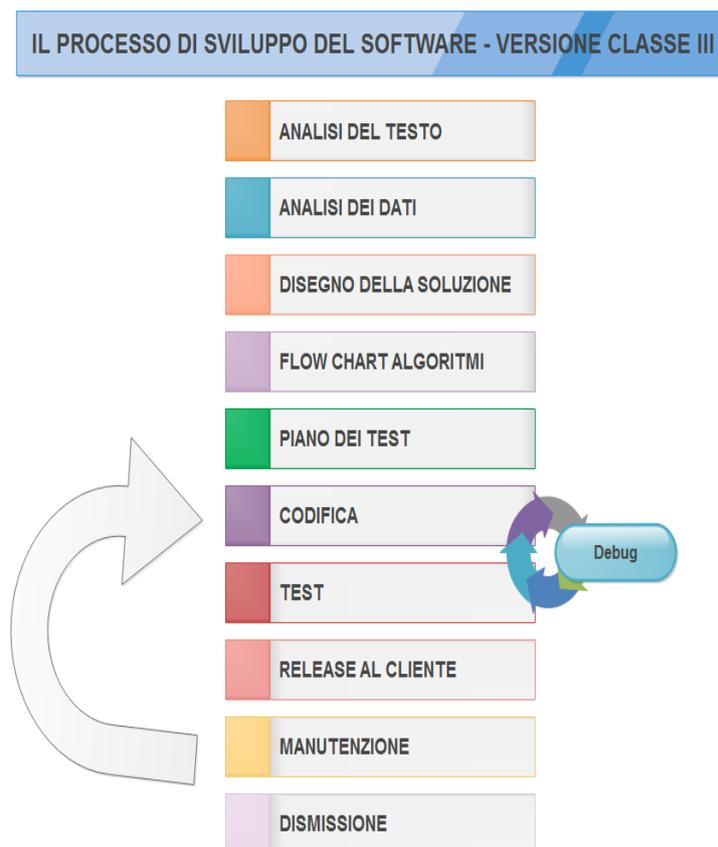
Il modello JAVA si è poi evoluto nella direzione con cui sono direttamente nati altri ambienti più recenti come .NET della Microsoft con i suoi linguaggi come C#: sull'elaboratore che prima interpretava l'IL è invece presente un compilatore che completa, ovviamente molto velocemente in quanto IL è già un 'semilavorato' come si diceva, la traduzione in linguaggio macchina. Questo ritardo, spesso impercettibile (perché un certo segmento di codice e solo quello viene tradotto solo nel momento in cui serve), lato utente è ripagato da una velocità di esecuzione molto simile alla forma compilata pura. Questo compilatore di codice IL è detto **compilatore JIT (Just In Time)** a sottolineare come la compilazione avvenga 'al volo', al momento e quando quel codice viene effettivamente invocato: il nostro programma potrebbe ad esempio prevedere una funzione per il calcolo dell'area di un pentagono ma la sua traduzione avverrebbe solo nel momento in cui per la prima volta quella funzione venisse richiesta.

## Dal problema al programma – un modello di sviluppo (semplificato) del software

Se vi chiedessi di scrivere un programma per il calcolo di una tassa iniziereste a scrivere il programma probabilmente quasi subito e lo potreste fare perché si tratta di un software minuscolo. Ma se vi chiedessi di scrivere un simulatore di aereo non sapreste neppure da che parte iniziare. Per un progetto del genere che richiede probabilmente un team di sviluppatori è necessario coordinarsi seguendo delle linee guida, un modello di conduzione del progetto che individui le fasi in cui scomporlo.

L'obiettivo dell'**ingegneria del software (software engineering)** è proprio quello di fornire modelli per guidare il lavoro del team di sviluppo. In quinta avrete una materia addirittura dedicata: Gestione progetto e impresa. Qui in terza ci acconteremo di un modello semplificato ma importante al nostro livello.

Il modello qui proposto, uno tra i tanti, è di tipo '**waterfall**' in quanto formato da fasi terminate una delle quali non si dovrebbe tornare indietro, proprio come l'acqua non può risalire una cascata (waterfall); chiusa una fase non ci si dovrebbe più preoccupare di essa. In realtà questa è una situazione ideale, teorica.



**Analisi del testo/problema**

Quante volte durante una verifica vi sono venuti dubbi su cosa intenda esattamente il prof. ?  
Quante volte il prof. ha aggiunto a voce ulteriori dettagli per chiarimenti? Ecco l'analisi di un testo dovrebbe far scaturire le stesse domande e osservazioni.

Si parte dalla richiesta per iscritto di un committente reale o dal testo di un esercizio scolastico e questi per quanto ci si sia sforzati di essere chiari conterranno spesso imprecisioni, ambiguità (un particolare che può essere variamente interpretato) e punti non sufficientemente definiti, casistiche non prese in considerazione ecc. L'obiettivo è rendere, si dice, ben posto il problema.

A seguire un esempio (si immagina che la richiesta ci sia stata inviata da un comune)

**TESTO ORIGINALE**

*Il nostro comune ha bisogno di una app sicura multiplatforma per il calcolo della spesa che periodicamente un alunno deve sostenere per trasporti per frequentare la scuola.*

Iniziamo con rileggere attentamente il testo individuando le parti poco chiare o incomplete/ambigue:

'... app sicura ...'

Il termine è troppo generico e rappresenta un cosiddetto requisito ambiguo; cosa significa qui esattamente? Che solo persone autorizzate possano usarla? (login con autenticazione), che nessuno possa intercettare i dati trasmessi? (uso del protocollo *https*), rendere i dati a prova di furto? (crittografia), tutte queste cose insieme?

'multiplatforma'

Altro requisito ambiguo: per quali piattaforme (Android, iOS ecc.)? Anche una sola piattaforma in più può fare una grossa differenza sui costi di sviluppo.

'... spesa che periodicamente ...'

Il termine periodicamente è ambiguo perché può avere diverse interpretazioni: si intende la spesa giornaliera? mensile o annuale? Non è sufficiente limitarsi a calcolare solo quella giornaliera e poi moltiplicare banalmente per 6 (o 5? potrebbe essere in vigore la 'settimana corta!') o per il numero di giorni medio di scuola mensile o annuale: staremmo infatti trascurando la possibilità di fruizione di abbonamenti settimanali, mensili od annuali che inciderebbero non poco sui costi

'... per trasporti ...'

Ambiguità. Qualunque mezzo di trasporto anche privato? O solo pubblici? Considerando solo strade a percorrenza gratuita o anche brevi tratti in autostrada con pagamento di pedaggi?

**Nota Bene**

Quando il progetto è reale ed esiste un vero committente una buona parte di questi dubbi saranno chiariti con il cliente.

Se si tratta del testo di un esercizio sarete voi a formulare delle cosiddette **ipotesi aggiuntive** (decidere voi come sciogliere i dubbi usando anche il buon senso e valutando **costi/benefici**).

A volte capita di individuare particolari sui quali un cliente vero non è in grado di intervenire; anche in questo caso saremo noi a completare con nostre ipotesi aggiuntive. Il problema deve in ogni caso essere ben posto.

## CHIARIMENTI CHE SI IMMAGINANO FORNITI DAL COMMITTENTE

E' il tipico caso in cui è opportuno consultare il committente (il comune). Le ambiguità e imcompletezze a seconda della interpretazione comportano costi di sviluppo diversissimi. Immaginiamo che il comune, contattato, precisi i seguenti aspetti:

- per *sicura* si intende che identifichi l'utente con il classico meccanismo del login (email + password)
- *multiplatforma*: **viene chiesto a noi di fare una valutazione e di scegliere**
- *periodicità*: mensile di default ma con la possibilità di variarla (giornaliera, settimanale, mensile, annuale, intero corso di studi; è in vigore la settimana corta (5 giorni di scuola)
- *mezzi di trasporto*: privati limitandosi a moto e auto; l'obiettivo è sensibilizzare la cittadinanza mostrando i risparmi che comporterebbe l'uso dei mezzi pubblici
- promozione dei mezzi pubblici
- sensibilizzazione ecologica

Siamo quindi chiamati a formulare una ipotesi aggiuntiva relativamente alle piattaforme da supportare: supportare piattaforme diverse da Android e iOS porterebbe a spese difficilmente giustificabili; infatti documentandoci sul web (motivare sempre le scelte!) si scopre che queste due piattaforme raccolgono più del 90% dell'utenza; in Europa e in Italia in particolare possiamo dire addirittura il 100%

Siamo finalmente pronti per riformulare il testo che si è arricchito di molte specificazioni che definiscono i cosiddetti **requisiti** cioè vincoli da rispettare, alcuni *funzionali* (funzionalità richieste al programma) ma altri potrebbero essere *non funzionali* (scelta del linguaggio da utilizzare, vincoli sui tempi di sviluppo, supporto di periferiche particolari, leggi da rispettare ecc.)

## RIFORMULAZIONE DEL TESTO

*Verrà sviluppata una app per sistemi operativi mobile Android e iOS per il calcolo della spesa mensile (20 giorni effettivi) che un alunno dovrebbe sostenere usando mezzi propri a motore per raggiungere il suo Istituto. L'accesso alla app sarà regolato da un login (email + password).*

*Saranno considerate diverse periodicità (giornaliera, settimanale, mensile, annuale, intero corso di studi) e fornito un confronto con tragitti alternativi che sfruttano mezzi pubblici evidenziando i risparmi conseguiti e gli impatti ecologici delle diverse soluzioni.*

## **Analisi Dei Dati**

Devono essere individuati i dati **indispensabili** da acquisire perché il programma non può calcolarli in autonomia. Tutti gli altri saranno determinati in automatico dal programma.

Facciamo un esempio: se ad un certo punto in un programma avessimo a disposizione una quantità espressa in ore e per proseguire fosse necessario esprimerla in secondi, sarebbe un grave errore richiedere che sia l'operatore umano ad inserire questo valore usando la tastiera costringendo lui a fare la conversione! Dovrà invece essere il programma a calcolare il valore richiesto.

Il secondo aspetto importante dell'analisi dei dati riguarda il dove memorizzarli (in grassetto le scelte tipiche per la classe terza):

- **array** e se vettori, vettori in parallelo o matrici
- **struct** e array di struct
- classi (ne parleremo in quarta!)
- **file**, definendone con precisione la struttura
- data base (ne parleremo in quinta!)

## **Disegno della soluzione**

Possiamo individuare due attività:

### 1. Definizione della struttura del programma

- per programmi semplici (centinaia di righe) scomporre ripetutamente il problema in altri più semplici fino ad individuare compiti che possano essere svolti da singole funzioni; una metodologia di scomposizione in funzioni molto diffusa è la **top/down** (vedi l'approfondimento sulla prossima pagina)
- per programmi standard (migliaia di righe e oltre) non si utilizza un solo sorgente ma si individuano diverse aree (ad esempio gestione interfaccia e interazione con l'utente, lettura/salvataggio dati, gestione della grafica o del suono, calcoli ecc.) per ciascuna delle quali si realizzano uno o più file sorgente contenenti funzioni e strutture dati; anche in questo caso si può usare la metodologia top/down

### 2. Definizione dell'interfaccia utente

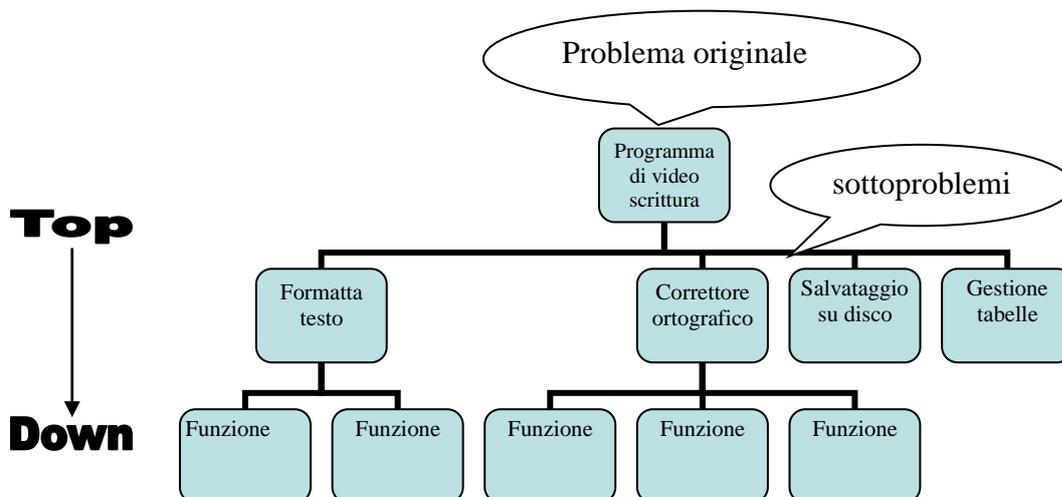
- progettare e disegnare le schermate con cui interagirà l'utente tenendo conto delle differenze di esperienza utente tra desktop, sito web, app mobile
- definire le modalità con cui l'utente interagirà con l'applicativo: solo tastiera, mouse? Touch screen? Comandi vocali ? ecc.

## Metodologia Top Down

Per problemi non banali, dal centinaio di righe di codice sorgente in su, scrivere un unico grosso blocco di codice nel main non è una soluzione accettabile. Anche per diminuire la difficoltà nel trovare una soluzione è molto meglio procedere ad una suddivisione in sottoproblemi. E' intuitivo che ogni sottoproblema sarà più semplice di quello originale. Per ogni sottoproblema che si ritiene ancora troppo complesso si procede ad una ulteriore suddivisione. Il procedimento viene ripetuto fino ad ottenere un certo numero di sottoproblemi sufficientemente semplici da essere risolti e codificati con una certa facilità, tipicamente scrivendo una funzione che a sua volta potrebbe invocarne altre.

NOTA: questa tecnica viene indicata anche come 'divide et impera' (dividi e mantieni sotto controllo) da un detto degli antichi romani: per mantenere sotto controllo un vasto territorio appena conquistato la popolazione veniva divisa e deportata in aree geografiche lontane tra loro. In questo modo quel popolo perdeva la sua unità e la sua forza ed era più facile dominarlo.

Questa scomposizione porta a realizzare una serie di **sottoprogrammi (subprograms, subroutines** cioè ) ciascuno dei quali risolve uno dei sottoproblemi terminali (quelli più semplici). Questa metodologia viene chiamata **TOP DOWN** (dall'alto al basso), perché graficamente possiamo rappresentarla come una struttura a piramide in cui in cima (top) si mette il problema originale e via via che si scende di livello sottoproblemi sempre più semplici, fino ad arrivare alla base (bottom) della piramide in cui troviamo i sottoproblemi più semplici di tutti.



Vediamo un altro esempio usando direttamente il codice invece di un diagramma. Obiettivo: un programma che permetta di giocare a dama. Inizialmente si rimane bloccati e non si sa da dove partire. La tecnica top down può ridurre questa complessità e portare a una soluzione passo passo. Si immagina che il gioco sia fatto partire dal main invocando una funzione principale:

```
int main () {
    gioca_a_Dama();
}
```

Poi si scompone il problema principale in sottoproblemi: all'inizio del gioco dovrà essere predisposta la scacchiera; poi con un ciclo che termina solo alla vincita di uno dei due giocatori (trascuriamo almeno per il momento il caso di partita patta):

```
gioca_a_Dama() {
    setupScacchiera();

    do {
        muove(giocatore1);
        muove(giocatore2);
    } while (!vince(giocatore1) && !vince(giocatore2) )
}
```

Certamente mancano ancora tanti dettagli ma il programma magicamente inizia a prendere forma! Notare come il controllo sulla vincita sia stato demandato a una ulteriore funzione che restituisce true/false per la condizione del ciclo.

In precedenza (analisi dei dati) avremo anche scelto come rappresentare un **giocatore** (con una struct), la **scacchiera** (una matrice di char: n=pedina nera, b=pedina bianca), una **mossa** ecc. potendo così dettagliare le invocazioni delle funzioni:

```
struct Giocatore { ... }
struct Mossa {... }

gioca_a_Dama() {
    Giocatore giocatore1 { ... }
    Giocatore giocatore2 { ... }
    char scacchiera[8][8];

    setupScacchiera(scacchiera, 8,8);

    do {
        muove(giocatore1, scacchiera);
        muove(giocatore2, scacchiera);
    } while (!vince(giocatore1, scacchiera) && !vince(giocatore2, scacchiera) )
}
```

E così via procedendo per raffinamenti successivi. Uno dei vantaggi di questa metodologia è che anche senza aver sviluppato davvero tutte le funzioni si può sperimentare con l'intero programma scrivendo dei cosiddetti "stub" cioè funzioni che si limitano a simulare il loro comportamento finale; ad esempio inserire le funzioni *setupScacchiera* e *vince* anche se non faremo fare loro niente la loro presenza è indispensabile per poter provare il resto del programma anticipatamente concentrandoci davvero solo su una parte delle altre che abbiamo deciso di sviluppare per prime;

```
void setupScacchiera(char scacchiera[][], int num_righe, int num_colonne)
{ //nessuna istruzione: è uno stub}

bool vince(Giocatore g, char scacchiera[][], int num_righe, int num_colonne)
{return false;} //restituisce un valore fisso di prova
```

## Flow Chart e Algoritmi

Terminata la parte di progettazione (analisi e disegno) ci avviciniamo alla scrittura del codice. Ogni algoritmo nuovo e non banale potrà essere rappresentato graficamente facilitando la sua realizzazione e la sua comprensione.

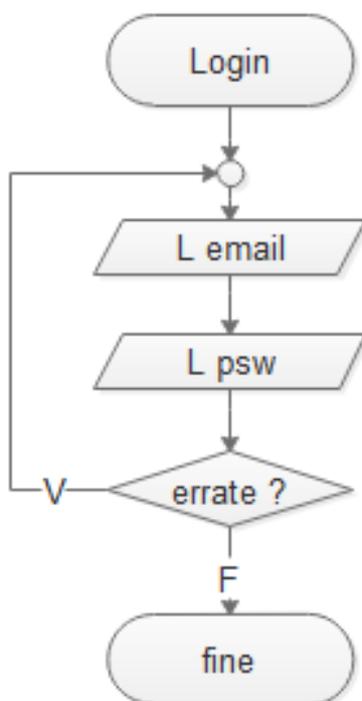
### Per le nostre attività scolastiche

Per programmi semplici, il flow chart coinciderà con l'intero programma e riprodurrà sempre lo schema: acquisizione dei dati, elaborazione, comunicazione dei risultati. Eventualmente questo schema sarà ripetuto all'interno di un ciclo. L'elaborazione sarà sufficientemente semplice da non richiedere un suo flow chart separato

Per programmi più complessi (quelli in cui saranno state individuate alcune funzioni) ci sarà:

- un flow chart che descrive il main
- un flow chart per ogni funzione che corrisponde ad una elaborazione non banale

**Per i dettagli sui simboli che possono essere usati con i flow chart e sulla loro composizione rimando all'ampia sezione che trovate più avanti nella dispensa;** quello che segue è solo un semplice esempio di flowchart.



Si inizia dall'alto dove un'etichetta funge da ingresso e descrive l'operazione: si tratta di un login che prevede l'inserimento di un indirizzo di email e una password.

Il pallino indica un punto a cui si può ritornare ciclicamente dal basso.

Si legge da tastiera l'indirizzo di email e a seguire la password.

Si testano le credenziali inserite; se è vero che sono errate (V) si ritorna all'inserimento di una nuova email; questo ciclo si ripeterà fino al momento in cui il test sarà falso (credenziali non errate).

Il login è terminato.

## Piano Dei Test

Potrà sembrare strano ma ancora prima di scrivere il codice bisognerebbe pensare a come verificare che produca risultati corretti. Per programmi semplici, come indicato qui sotto, è sufficiente pensare a combinazioni sia tipiche sia critiche di valori da fornire come input da tastiera verificando i risultati (lo zero, valori negativi, stringhe vuote, valori molto grandi o molto piccolo sono di solito buoni candidati per il test).

Per programmi complessi dovremo comprendere anche file da leggere, dati provenienti da una connessione di rete ecc. individuando anche in questo caso set di valori con cui mettere alla prova il software; è necessario simulare anche situazioni critiche di altro tipo: cosa succede se il file non esiste o non contiene dati, se ne contiene di più di quelli che il programma può gestire (array pieni ad esempio), se il file non viene trovato, se la connessione di rete non può essere stabilita o se il pacchetto dati ricevuto è corrotto ecc.

All'atto pratico, **per i nostri programmi a livello scolastico, è sufficiente provare** il programma (vedi la fase di test più avanti) tenendo sotto mano lo schema qui sotto usandolo come 'scaletta' senza scrivere in anticipo un vero e proprio piano dei test.

Sostanzialmente si tratta di individuare anticipatamente degli insiemi di input con cui sarà messo alla prova il programma prima a livello di ogni singola funzione (unit test) e poi tutto insieme (integration test); i dati usati come input per il test devono comprendere delle scelte di valori 'normali' ma anche tutte le combinazioni di valori 'critici', cioè che violano i vincoli stabiliti, che si possono presentare. Un programma solido anche in presenza di combinazioni che non consentono di proseguire dovrebbe segnalare le situazioni di errore e consentire la continuazione senza bloccarsi o addirittura bloccare (si parla di crash) il sistema su cui è in esecuzione

- **per variabili numeriche** provare valori:

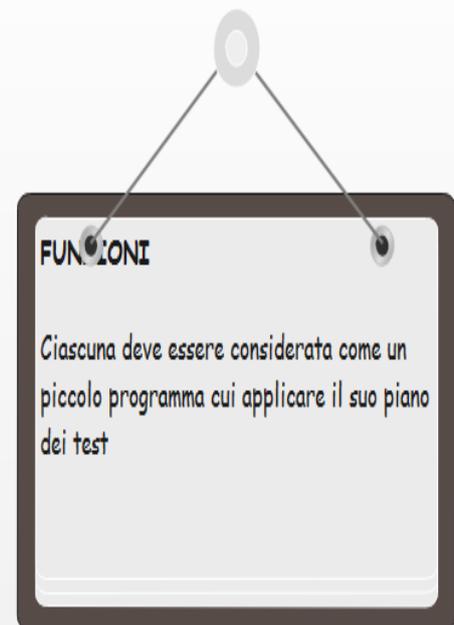
- \* nulli, cioè saltare l'inserimento quando richiesto
- \* negativi
- \* lo zero
- \* al di fuori dell'intervallo accettabile o dell'insieme accettabile (vedi la parte di analisi dati)
- \* estremamente piccoli (vicino allo zero) o estremamente grandi (sia negativi sia positivi)

- **per variabili carattere e stringhe** provare con valori:

- \* nulli, cioè saltare l'inserimento quando richiesto
- \* stringa nulla ""
- \* maiuscole / minuscole
- \* formate da una sola parola o da più parole
- \* con spazi all'inizio e/o alla fine

- **per files:**

- \* con file vuoto
- \* con file mancante di alcune parti previste nella struttura (vedi la parte di analisi dati)
- \* inserendo nel file le stesse combinazioni di valori critici visti ai due punti precedenti



Aggiungo qui una parte per gli array: verificare il codice che li usa in caso di array vuoto o pieno

**Codifica**

Mentre le fasi precedenti sono compito dell'**analista** le successive sono il tipico campo di azione del **programmatore**. Spesso un programmatore con esperienza svolge entrambi i ruoli comportandosi da analista / programmatore; stessa situazione per programmi semplici come quelli che svilupperemo a scuola.

Nella fase di codifica si scrive il codice sorgente (le istruzioni del programma nel linguaggio di programmazione scelto) usando come linee guida tutti i documenti prodotti nelle fasi precedenti (flow chart in primis).

Il programmatore insomma segue le linee guida individuate dall'analista e deve tradurre il progetto in programmi funzionanti. E' comunque una fase che lascia spazio all'iniziativa personale ed anche alla creatività del programmatore nel trovare le migliori tecniche per raggiungere un certo obiettivo. Ad esempio nel mettere in ordine crescente o decrescente gli elementi di un array esistono decine di algoritmi ciascuno da preferire a seconda della situazione e un bravo programmatore saprà in modo motivato quale usare.

Entriamo ora nel dettaglio della fase di codifica, definendo la cosiddetta **catena della programmazione** (cioè la sequenza delle sue sotto fasi).

**(A)** Il programmatore inizia con lo scrivere le istruzioni (nel linguaggio di programmazione scelto) con un programma chiamato **editor**. Questo termine inglese significa *redattore* cioè colui che cura la scrittura di un testo. Molto semplicemente, pensate all'editor come ad un programma di videoscrittura specializzato per la scrittura di un programma informatico. Il file scritto con l'editor prende il nome di **codice sorgente** (**source code**).

Oggi gli editor sono diventati molto sofisticati e ricchi di funzionalità e si preferisce parlare di **IDE, Integrated Development Environment (ambienti integrati di sviluppo)**.

I file dei codici sorgente vengono registrati sul disco con un'estensione (la parte del nome del file dopo il punto) che aiuta a riconoscere il linguaggio di programmazione utilizzato. Ecco le principali estensioni valide per noi: .c (C), .cpp (c++), .asm (*assembly*), .cs (C#), .java (JAVA), .php (PHP), .js (Javascript). Pur non trattandosi di file di istruzioni eseguibili vale la pena ricordare anche: .html (pagine WEB), .css (fogli di stile per le pagine WEB).

Caratteristiche principali di un moderno editor integrato in un IDE:

- modelli (template) di progetto predefiniti (modello game, modello sito web, app ecc.)
- autocompletamento delle istruzioni e dei nomi delle variabili
- colorazione del codice (syntax highlighting): colori diversi per keyword, identificatori di variabili, commenti, stringhe ecc.
- folding/unfolding: comprimere o espandere blocchi di codice ad esempio mantenendo di una funzione visibile solo l'intestazione con un simbolo '+' per espandere e '-' per comprimerer
- suggerimenti dei parametri delle funzioni alla loro invocazione
- bilanciamento delle parentesi ed evidenziazione aperte / chiuse
- evidenziazione in tempo reale degli errori lessicali e di sintassi
- refactoring del codice: ristrutturazione intelligente del codice; ad esempio accurata ricerca e sostituzione di un identificatore di variabile con un altro distinguendo tra array, variabili semplici, commenti, contenuto di costanti stringa ecc

Altri strumenti notevoli

- *debugger (in assoluto il più utile)*: offre la possibilità di stabilire uno o più punti di interruzione (break point) e di eseguire in modalità debug il programma; in tale modalità l'esecuzione si blocca sulla riga di un breakpoint dando la possibilità di investigare il contenuto di variabili, array e altre strutture dati, i registri della CPU, la sequenza (stack) delle chiamate delle funzioni, osservare il codice assembly generato dal compilatore e altro; al break point il programmatore può far avanzare il programma sotto il suo controllo in diversi modi: alla prossima istruzione, saltando o entrando in dettaglio in una funzione, fino alla riga su cui si posiziona il cursore; un break point può essere condizionale (esecuzione bloccata solo se vera una certa condizione specificata; ad esempio: fermati su questa riga SOLO SE il valore della variabile x diventa negativo; oppure SOLO SE è la centesima volta che esegui questa istruzione (utile con i cicli e gli array)

- *profiler (profilatore)*: fornisce una mappa della percentuale di tempo usata da ciascuna parte del programma; utilissimo per l'ottimizzazione delle prestazioni: potremmo ad esempio scoprire che il 50% dell'intero tempo di esecuzione il programma lo spende eseguendo una sola delle molte funzioni presenti e questa diventerebbe oggetto privilegiato dei nostri sforzi di miglioramento (anche un modesto miglioramento peserebbe molto sull'intero programma); viceversa tentare di ottimizzare una funzione in cui viene speso lo 0.01% del tempo contribuirebbe poco anche raddoppiando le sue prestazioni

- generatore automatico di documentazione (se avremo inserito seguendo una precisa convenzione adeguati commenti nel codice)

- gestione delle versioni del codice (*versioning*): spesso si lavora in team ed è difficile tenere traccia delle modifiche apportate nel tempo e da chi; un sistema di versioning è in grado di ripristinare qualunque vecchia versione del progetto e di tenere coordinate le modifiche di tutti i membri del team; tra i migliori sistemi di versioning on line ricordiamo almeno GitHub (ora acquisito da Microsoft ma sempre gratuito).

**(B)** Il **compilatore** trasforma il codice sorgente nel cosiddetto **codice oggetto** (**object code**) i cui files hanno di solito estensione *.obj*; sarebbe più corretto dire che il compilatore *tenta* la trasformazione perché questo processo di traduzione potrebbe interrompersi per vari motivi:

- **Errori lessicali (lexical errors)**. Viene scoperto un simbolo che non appartiene al cosiddetto dizionario delle parole chiave (**keyword**) proprie del linguaggio e neppure all'insieme degli identificatori (nomi variabili, funzioni, costanti ecc.) aggiunti dal programmatore: corrisponde di solito ad un errore di battitura. Gli editor di oggi segnalano questi errori durante la digitazione senza aspettare l'intervento del compilatore.
- **Errori sintattici (syntax errors)**. Anche se viene superata la fase di analisi lessicale ci potrebbero essere ancora errori di tipo sintattico, cioè che violano una regola sintattica di quel linguaggio (ad esempio dimenticare il punto e virgola alla fine di una istruzione C++). Anche in questo caso gli editor moderni segnalano in anticipo la maggior parte dei questi errori.

- **Errori semantici:** sono quelli legati al significato delle istruzioni coinvolte ed al tipo dei valori che queste stanno elaborando; ad esempio l'istruzione di assegnamento  $x = 3 + \text{"ciao"}$ ; dove  $x$  è stata dichiarata di tipo *int*, è corretta da un punto di vista lessicale e sintattico. Ma se consideriamo l'aspetto del significato di ciò che stiamo usando ci accorgiamo che non possiamo dare un significato alla somma di un numero e una stringa di caratteri per generare come risultato un numero; stiamo usando tipi incompatibili anche in relazione all'operazione comandata. Un'altra situazione in cui si evidenziano errori semantici è quella in cui viene invocato un sottoprogramma fornendo parametri del tipo sbagliato o valori non ammessi, come in `sqrt( 'errore madornale!' )` perché ovviamente non è possibile calcolare la radice quadrata di una stringa, o in `sqrt( -23 )` perché l'operazione non è definita per argomenti negativi.

NOTA: questi primi tre tipi di errore sono quasi tutti intercettati dal compilatore e per questo sono detti a `compile time` cioè prima che il programma sia mandato in esecuzione.

- **Errori logici:** sono i più difficili e pericolosi; si evidenziano a `run time` cioè durante l'esecuzione del programma. Il compilatore non se ne può accorgere purtroppo; ad esempio una formula con un divisore che può in certe situazioni diventare zero; oppure un argomento di radice quadrata che può diventare negativo; oppure aprire un file, dimenticarsi di chiuderlo e in alcune situazioni tentare di riaprirlo; iniziare ad elaborare gli elementi di un array dall'indice 1 e non 0 (oppure superare l'ultima posizione valida). Da notare che il caso in cui si evidenziano è il più fortunato: molto peggio non accorgersene ed usare dei dati errati o lasciare una vera e propria bomba ad orologeria nel codice che potrebbe scoppiare nel momento meno opportuno per il cliente! Ecco perché è fondamentale svolgere bene il test di un programma (vedi più avanti la sezione specifica sul test)

**(C) Il linker.** Il codice oggetto, risultato della compilazione, non sempre è del tutto tradotto al 100% in codice binario eseguibile dalla CPU. Questo per due motivi:

a) Ogni programma non banale è di solito scritto usando più di un file sorgente ma molti compilatori (tra cui quelli del C++) quando traducono un file sorgente non tengono conto degli altri anche se degli altri in quel sorgente si sta usando una funzione o una variabile; quando allora in un file sorgente il programmatore usa una variabile o chiama una funzione che si trova in un altro il compilatore lascia queste parti in sospenso perché non ha tutti gli elementi necessari (traduzione incompleta).

### Nota

Questa modalità di condurre la compilazione ha almeno due grossi vantaggi:

- Essendo la compilazione di un sorgente indipendente da quella degli altri si possono compilare in contemporanea (anche su elaboratori diversi o semplicemente su processori multi core) più file sorgente per volta abbreviando tremendamente i tempi di compilazione del progetto
- Se si compie una modifica a un file sorgente potrà essere ricompilato solo quello di nuovo con grande risparmio sui tempi di compilazione dell'intero progetto

b) Molto spesso il programmatore invoca funzioni (pensate alla *sqrt* per calcolare la radice quadrata) che sono state raccolte in librerie (libraries) che altro non sono che sorgenti precompilati per risparmiare tempo perché si tratta di comandi usati assai di frequente che in questo modo non dovranno essere ricompilati tutte le volte insieme ai nostri: questo codice binario dovrà ovviamente essere incorporato con la traduzione del nostro ma questa operazione non viene svolta dal compilatore perché come già detto lavora solo sul file sorgente che sta traducendo in un certo momento. **Nota:** anche noi potremmo trasformare un file sorgente contenente funzioni in una libreria.

Il **linker** è appunto il programma che prendendo in rassegna tutti i codici oggetto (*obj*) risultato delle traduzioni dei nostri file sorgente ed estraendo solo il necessario (altro vantaggio) dalle diverse librerie deve fondere il tutto creando l'eseguibile (.exe)

**Linee guida stilistiche per la codifica (OBBLIGATORIE!)**

Rispettare le seguenti linee guida (per programmi non banali)

- **intestazione programma** (come commento ovviamente); a seguire uno schema che potete riprodurre in ogni vostro sorgente

```
/* (descrizione) Questo programma accetta in input i path ed i nomi di due file di testo
e stampa a video le righe che sono diverse
```

```
Sviluppato da: Rossi Mario rossi.mario@rossim.it
Versione attuale: 2.3 Release iniziale: 07-01-2015
```

History (dalla più recente alla meno recente)

2.3 31-7-2016

- Risolto il bug con crash per nomi di files più lunghi di 127 caratteri
- Migliorate le performance del 30% con file più grandi di 128MB

2.2 23-7-2016

- Eliminati diversi piccoli bug
- Ora il programma riconosce la codifica UTF-8

ecc. ecc.

```
*/
```

- **per la scrittura di codice in generale:**

\* spaziare secondo logica (tenere unito il codice che è correlato e spaziarlo dal resto)

```
//questa parte riguarda il triangolo
int base=0;
int altezza=0;
```

```
//questa parte riguarda il cerchio
int raggio=0, int centro_x=0, int centro_y=0;
```

```
//questa parte riguarda tutte le figure geometriche
//ma nessuna in particolare
double perimetro=0;
double area=0;
```

\* indentare in modo coerente → → → → → →

**Dopo l'apertura di una graffa di un if, un else o di un ciclo rientrare di due spazi e allineare in quel punto tutte le istruzioni dentro quel blocco. La parentesi graffa di chiusura allineata con quella di apertura.**

```
if (...)
{
...
for (...)
{
...
if (...)
{
...
}
else
{
...
}
} //for
} //if
```

Possibili stili di indentazione (notare come ogni indentazione usi sempre lo stesso numero di spazi di rientro; suggerisco 2 spazi, valore configurabile negli editor)

Esempio 1 (personalmente preferisco questo stile che ritengo più chiaro)

```
for(...)
{
  int tot=0;
  if (...)
  {
    istruzione;
    while (...)
    {
      istruzione;
      istruzione;
    } //graffa chiusura allineata con apertura
  } //idem
} //idem
```

Esempio 2 (risparmio di una riga per la graffa di apertura rispetto al primo esempio)

```
for(...) {
  int tot=0;
  if (...) {
    istruzione;
    while (...) {
      istruzione;
      istruzione;
    } //graffa fine while allineata con la 'w' di while
  } //allineata con inizio if
} //allineata con inizio for
```

\* una istruzione sola per riga

## Test

Terminata la scrittura del programma (o di parte di esso) inizia la fase di test. Sottoporre a test un programma significa provarlo con una scelta sufficientemente ampia di combinazioni di dati in *input* normali ed altri 'limite'. **Questi dovrebbero essere già stati individuati nella fase di pianificazione dei test vista in precedenza.** Facciamo un semplice esempio immaginando di avere scritto un programma che forniti due numeri in *input* calcola che percentuale è il primo rispetto al secondo; quindi se il primo numero fosse 50 ed il secondo 150 il risultato fornito dovrebbe essere 33,3 % periodico (50 è infatti un terzo di 150...). Non è difficile convincersi che nel programma la formula risolutiva è:

$$(\text{primo numero}/\text{secondo numero})*100$$

Fare il test di questo programma con configurazioni di dati in *input* 'normali' significa provare il programma con coppie di numeri tipo (10,20) (50,150) ecc. Poi ci si potrebbe domandare se il programma fornisce risultati corretti anche quando il primo numero è maggior dal secondo: (240,60); e scopriremmo che la risposta è sì: otterremmo come valore 400% (in effetti, 240 è il quadruplo di 60). E se usassimo numeri negativi? Nessun problema...

Ok, è arrivato il momento di essere cattivi: e se usassimo numeri decimali? Tipo (10.2, 97.5) ? E se il primo numero fosse zero come in (0, 34)? Anche con queste configurazioni di valori in *input* il programma continua a fornire risultati corretti. Giunti a questo punto il programmatore inesperto (o pigro) potrebbe concludere che il programma funziona bene in tutti i casi possibili. Purtroppo la matematica c'insegna che non è possibile dividere per zero: inserendo una configurazione di *input* con il secondo valore uguale a zero, come in (72,0) il programma andrebbe letteralmente in tilt! Gli informatici in questi casi usano un'espressione assai colorita: il programma va in **crash** ! (un crollo, uno schianto al suolo).

Il caso dello zero come secondo numero è un cosiddetto **caso limite**: ogni programma dovrebbe essere testato in tutti i casi limite che potrebbero presentarsi (anche quelli con probabilità molto bassa!).

La fase di test, come già discusso in precedenza, ha lo scopo di evidenziare gli errori durante il funzionamento del programma, i cosiddetti '**run time error**'. Alcuni di questi sono legati all'uso di risorse hardware o all'interazione con il sistema operativo (un file che non viene trovato, una periferica spenta o in avaria, il quantitativo di RAM richiesta che non è disponibile ecc.); altri sono veri e propri **errori logici**. Questi sono i più difficili da evidenziare; il programmatore ha rispettato tutte le regole del linguaggio ma ha sbagliato ad impostare la soluzione! Il compilatore non può fare nulla ...

```
if (x>=0)
  cout << "Il valore della x è negativo";
else
  cout << "Il valore della x è positivo";
```

La logica è chiaramente invertita: per il compilatore invece questo segmento di codice è **perfetto** !

Il test fatto dai programmatori stessi o da personale interno alla ditta che sviluppa il software (**software house**) è chiamato **alfa test** e si parla di versioni alfa dell'applicazione. Successivamente si può rilasciare gratuitamente in prova ad utenti esterni (interessati per vari motivi; a voi non piacerebbe ad esempio testare in anteprima un video game famoso??): si parla di **beta test** e di versioni beta dell'applicazione.

## **Debug**

La fase precedente ha il solo scopo di evidenziare la presenza di un errore. La fase di debug indica invece l'attività del programmatore volta a isolare le cause (la porzione di codice responsabile) e l'eliminazione (modifica del codice) di un errore di tipo **LOGICO** (seguono alcuni esempi; altri li avevo indicati parlando delle tipologie di errore)

- una variabile non inizializzata con il giusto valore
- un contatore che ci si è dimenticati di incrementare o che è incrementato di una quantità sbagliata
- non aver controllato in una divisione senza che il denominatore sia diverso da zero
- usare una funzione con parametri con valori inaccettabili (sqrt di un numero negativo)
- una condizione di un if/for/while sbagliata: && invece di || o viceversa, & o | invece di && e ||, < invece di > ecc.
- non aver messo le parentesi graffe ad un if, for, while quando c'è più di una istruzione
- aver dimenticato un case in uno switch
- logica stessa dell'algoritmo sbagliata (il più difficile da scoprire)

*ATTENZIONE!! Una delle cattive abitudini più comuni è quella di non testare nuovamente il programma dopo una modifica. Purtroppo l'esperienza insegna che, non così raramente come si potrebbe pensare, le modifiche apportate per correggere un errore ne introducono altri, a volte peggiori!*

## Strumenti per il debug

Sono sostanzialmente il debug manuale con **trace table** e quello supportato da un apposito strumento software, il **debugger**; ricorderete che di quest'ultimo già parlato nella sezione sugli IDE per cui ora sarà descritto solo il primo.

**Tabella di traccia (Trace Table).** L'idea è quella di mettersi nei panni del *microprocessore* 'eseguendo' il programma istruzione per istruzione ma con carta e penna (è anche un ottimo esercizio per migliorarsi come programmatori alle prime armi). Si tratta di tenere traccia in una tabella di ogni modifica ai valori delle variabili, una istruzione alla volta. Spesso ci si accorge di un errore perché una variabile ad un certo punto dell'esecuzione assume un valore inatteso (o nessun valore).

Qui a seguire un esempio per un ciclo che calcola la somma dei numeri da 1 a 4 (1+2+3+4).

```

1 int n=1, somma=0;
2 while (n<4)
    {
3   somma += n;
4   n++;
    }
5 cout << somma;
    
```

Le variabili da tenere sotto controllo sono 2: il contatore **n** e **somma**; per seguire meglio l'esecuzione le righe del programma sono state numerate da 1 a 5 ed ogni volta che si immagina di eseguire una istruzione si aggiunge una riga alla tabella con il numero di quella istruzione e su quella stessa riga si indicano i valori delle variabili eventualmente modificati per effetto dell'esecuzione delle istruzioni. Da valori inaspettati delle variabili di solito si riesce a capire qual è la parte del programma che causa l'errore.

Istruzione n.	<b>n</b>	<b>somma</b>
<b>1</b>	1	0
<b>2</b>	1	0
<b>3</b>	1	1
<b>4</b>	2	1
<b>2</b>	2	1
<b>3</b>	2	3
<b>4</b>	3	3
<b>2</b>	3	3
<b>3</b>	3	6
<b>4</b>	4	6
<b>2</b>	4	6

condizione while: true

condizione while: true

condizione while: true

condizione while: false !!!! **errore !!!** Ci accorgiamo che il ciclo while termina prima di aver sommato il numero 4

**Release**

Il programma è ora ben testato, gli errori eliminati. A questo punto se il programma ci fosse stato commissionato da un cliente saremmo pronti per la consegna (si parla di **release**, cioè rilascio, del software). A seconda del tipo di applicazione potrebbe essere sufficiente spedire il software al cliente che provvederà personalmente ad installarlo. In casi più complessi potrebbe invece rendersi necessario recarsi presso il cliente e provvedere all'installazione, collaudo, formazione del personale.

**Manutenzione**

La Inizia poi una fase lunga e costosa chiamata **manutenzione (maintenance)**. Sono le modifiche che ci verranno chieste di apportare al programma per errori trovati durante l'utilizzo reale dal cliente e sfuggiti al nostro test (**manutenzione correttiva**), oppure per migliorarlo rendendolo più veloce nel fornire risultati, fargli usare meno memoria, aggiungere funzionalità ecc. (**manutenzione perfetta**) o infine per adattarlo a nuove situazioni come una nuova legge cui adeguarsi o una modifica ad una legge esistente o procedure di lavoro presso il cliente modificate (**manutenzione adattativa o adattiva**).

**Dismissione**

Infine quando un'applicazione non è più utile (perché superata da altre, perché le modifiche richieste sono troppe e non economicamente sostenibili, perché un evento imprevisto la rende obsoleta ecc.) se ne può anche decretare la morte, la dismissione.

**Algoritmi (algorithms)**

Un **algoritmo** è la descrizione *non ambigua, effettivamente eseguibile e deterministica* delle istruzioni da eseguire in un *numero finito di passi* per risolvere *una intera classe di problemi*.

*Caratteristiche di un algoritmo in dettaglio*

**Non ambiguo:** un algoritmo deve essere composto solo da istruzioni non ambigue: l'istruzione 'ripetere per un po' di volte' è ambigua; 'ripetere l'istruzione per 10 volte' non è ambigua. E' proprio grazie al rigore dei linguaggi di programmazione che vengono eliminate le ambiguità.

**Effettivamente eseguibile.** In un algoritmo si devono usare solo istruzioni che l'esecutore è effettivamente in grado di compiere. Se un esecutore non è capace in modo autonomo di calcolare la radice cubica di un numero dovremo essere noi a fornire una tecnica di calcolo che usi solo le operazioni aritmetiche note all'esecutore.

**Finito.** Un algoritmo deve terminare usando un numero finito di passi.

**Deterministico.** a fronte delle stesse condizioni iniziali e dati forniti in ingresso deve produrre sempre gli stessi risultati.

**Generale.** Un algoritmo non è tale se risolve solo un caso particolare di un problema: deve essere utile per la soluzione di un'intera classe di problemi. Ad esempio, un algoritmo non può servire a calcolare la radice quadrata di un numero particolare ma di un numero qualsiasi.

E' molto importante distinguere il **risolutore**, cioè l'ideatore di un algoritmo, dall'**esecutore**. Al risolutore sono richieste molte capacità e vera intelligenza; l'esecutore deve solo essere in grado di eseguire meccanicamente, anche senza capirle, le istruzioni dell'algoritmo.

**Il computer è solo un esecutore** incredibilmente veloce e dotato di memoria perfetta a cui è sufficiente fornire descrizioni di algoritmi (programmi)

Ma come può essere concretamente descritto un algoritmo? Arrivati a questo punto dobbiamo confrontarci con l'assoluta inadeguatezza del linguaggio parlato (il cosiddetto **linguaggio naturale, natural language**).

Banalizzo con un classico esempio:

*la vecchia porta la sbarra*

Quale significato deve essere dato a questa frase? Si tratta forse di un'anziana signora china sotto il peso di una pesante sbarra? O si sta parlando di una uscita sbarrata da una vecchia porta? Questa ambiguità è inaccettabile per un microprocessore: esso deve sapere esattamente come comportarsi.

Per descrivere un algoritmo ricordiamo almeno questi strumenti:

1. testuali: **pseudo codice** (pseudo code)
2. grafici: i **diagrammi di flusso** (**flow chart**)

Lo pseudo codice potrebbe essere definito come un linguaggio di programmazione semplificato (e volendo con istruzioni in italiano) in quanto consente di trascurare molti dettagli che invece dovranno essere presi in considerazione scrivendo il programma vero e proprio; lo pseudocodice  coglie invece gli aspetti **essenziali** di un algoritmo.

Ecco come potrebbe apparire lo pseudo codice per la stampa dei numeri interi nell'intervallo [A, B] con A e B introdotti da tastiera:

```
leggi A, B
se A>B allora
  scambia i valori di A e B
ripeti
  scrivi A
  aggiungi 1 ad A
finchè A<B
```

NOTA: niente messaggi per l'utente tipo: *Inserire l'estremo inferiore dell'intervallo* e poi *Inserire ... superiore ...* perché non sono indispensabili per descrivere l'algoritmo.

Certe operazioni sono in realtà delle macro operazioni: molti linguaggi di programmazione non avrebbero singole istruzioni per scambiare i valori di A e B. Ma il significato dell'operazione è sufficientemente chiaro per un programmatore che provvederà ad ottenere lo scambio con una sequenza di istruzioni equivalenti ed effettivamente a disposizione nel linguaggio di programmazione in uso.

---

**Di seguito esamineremo i comandi di base e le strutture fondamentali della programmazione mostrando come sono rappresentate con i flow chart e lo pseudo codice; il vostro professore sceglierà poi quale forma usare.**

---

## I flow chart e le tre strutture fondamentali della programmazione

---

I *flow chart* sono diagrammi (disegni tecnici) che descrivono visivamente un algoritmo. Un loro punto di forza, come per lo pseudocodice, è il non essere legati ad uno specifico linguaggio: il *flow chart* è un linguaggio visivo universale e non è legato a nessun linguaggio di programmazione specifico; **ogni programmatore non potrà che interpretare allo stesso modo un qualunque flow chart** (meno vero per lo pseudocodice che è legato a una lingua)

Per costruire un flow chart si usano dei simboli che rappresentano istruzioni di solito disponibili nei vari linguaggi di programmazione. Questi simboli devono essere connessi tra loro a rappresentare il flusso di esecuzione che seguirà l'esecutore. Questo deve avvenire con una tecnica di costruzione rigorosa e ordinata diversamente otterremmo dei programmi difficilmente comprensibili e difficilmente modificabili.

Nel 1966 due informatici, Jacopini e Böhm, dimostrarono che qualsiasi programma poteva essere scritto in modo ordinato usando solo tre strutture (modi di organizzare le istruzioni) fondamentali: **sequenza, selezione ed iterazione**. Si parla quindi di **programmazione strutturata**. Questo concetto è immediatamente applicabile anche ai flow chart:

**La sequenza** corrisponde banalmente alla possibilità di far eseguire le istruzioni una dopo l'altra nell'ordine in cui sono scritte leggendo il diagramma dall'alto verso il basso.

**La selezione** che corrisponde alla possibilità di scegliere di eseguire uno solo tra due blocchi di istruzioni a seconda del verificarsi o meno di una *condizione* (ad esempio: se l'età è maggiore di 18 allora esegui queste istruzioni altrimenti esegui queste altre).

**L'iterazione** corrisponde invece alla possibilità di far ripetere un blocco di istruzioni fintanto che una certa condizione rimane vera; quando la condizione diventa falsa la ripetizione termina.

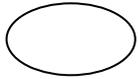
Esistono anche simboli speciali (inizio programma, fine programma ecc.) che non rappresentano istruzioni vere e proprie ma che sono utili per la costruzione del *flow chart*.

I *flow chart* consentono (se usati adeguatamente) di catturare **l'essenza** (cioè le cose importanti) di un algoritmo **senza curarsi di particolari irrilevanti** per la soluzione del problema.

Ad esempio, pensando ad un programma per la soluzione delle equazioni di secondo grado, non ha senso considerare nel *flow chart* i colori con cui verranno visualizzati i messaggi o i messaggi stessi che saranno proposti all'utente: con un tale esagerato livello di dettaglio, infatti, tanto varrebbe scrivere direttamente il codice del programma finale!

## I principali simboli usati nei flow chart

### **Inizio / Fine programma**



Spesso vista la semplicità dei nostri flow chart lo si omette

**Versione pseudo codice:** *Inizio* Anche in questo caso non lo si mette quasi mai

### **Assegnamento di un valore ad un simbolo (variabili)**

età ← 18    la variabile *età* assume il valore 18

**Versione pseudo codice:** età ← 18

**Scrittura di un valore sul video** o periferiche di output. Ad esempio la scrittura della scritta "ciao" (le virgolette servono a distinguere le scritte dai nomi delle variabili)



Suggerimento mnemonico: questo simbolo ricorda il profilo di un vecchio monitor non LCD



### **NOTA**

Per la stampa a video esiste un simbolo alternativo molto usato (anche nelle nostre soluzioni già pubblicate on line o inserite nelle dispense di laboratorio):



La **S** ci ricorda che è una **Scrittura** (oppure **O** = **Output**) perché lo stesso simbolo può essere usato per la lettura di un dato dalla tastiera (vedi sotto)

**Versione pseudo codice:** scrivi "ciao"

**Acquisizione di un valore tramite tastiera** (o altre periferiche di input) con sua memorizzazione nella variabile indicata . Esempio: inserire con la tastiera dove abita una persona e memorizzare l'informazione nella **variabile** *indirizzo*.

Una **variabile** (come verrà spiegato meglio nella sezione sul c++) è un nome che rappresenta un valore memorizzato da qualche parte nella memoria del computer; tramite questo identificatore potremo recuperare o modificare (da cui il nome 'variabile') quel valore.



Suggerimento mnemonico: questo simbolo ricorda il profilo di una tastiera vista di fianco



**NOTA**

Per la stampa a video esiste un simbolo alternativo molto usato (anche nelle nostre soluzioni già pubblicate on line o inserite nelle dispense di laboratorio):



monitor (vedi sopra)

La **L** ci ricorda che è una **L**ettura (oppure **I** = **I**ntput) perché lo stesso simbolo viene può essere usato per la scrittura di un dato sul

**Versione pseudo codice:** leggi indirizzo

**ESEMPI (in blu la versione pseudo codice, identica in questo caso)**

$$x \leftarrow y$$

dai ad x lo stesso valore di y

$$x \leftarrow y$$

$$x \leftarrow y - 1$$

dai ad x il valore di y diminuito di 1

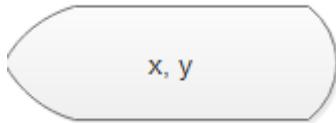
$$x \leftarrow y - 1$$

$$x \leftarrow (5 - 2) * (4 + 3)$$

dai ad x il valore dell'espressione  $(5 - 2) * (4 + 3)$

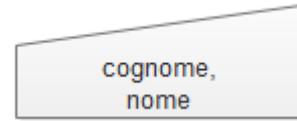
$$x \leftarrow (5 - 2) * (4 + 3)$$

Per praticità si possono specificare anche più dati da leggere / scrivere per volta:



scrivi sul video prima il valore della variabile *x* e poi quello della variabile *y*

*scrivi x, y*



accetta dalla tastiera un primo valore che memorizzerai nella variabile *cognome* e poi un secondo valore che memorizzerai nella variabile *nome*

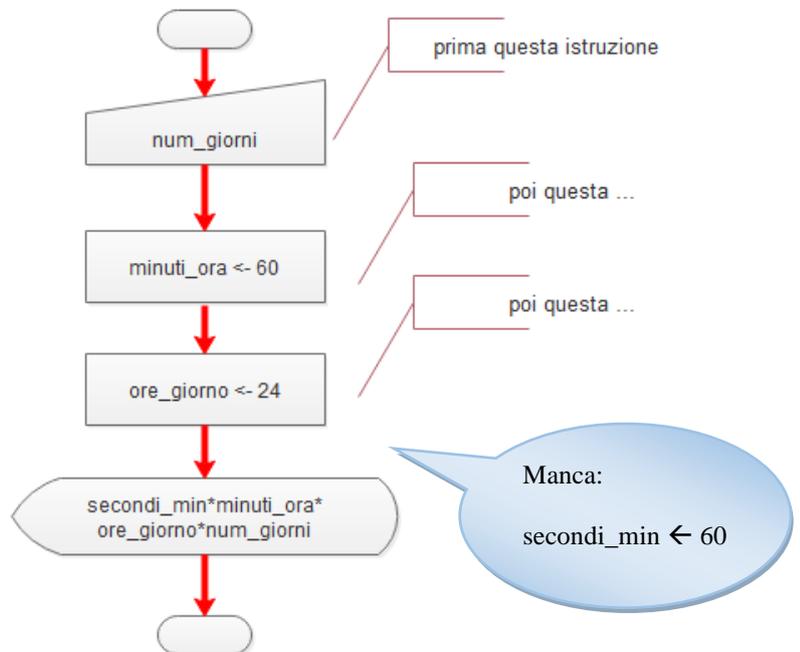
*leggi cognome, nome*

*I simboli visti fino ad ora servono per istruzioni singole. Vediamo ora come si rappresentano le strutture fondamentali della programmazione (sequenza, selezione, iterazione).*

### Sequenza

Per indicare che due istruzioni vanno eseguite una dopo l'altra si mettono i loro simboli uno sotto l'altro collegandoli con una freccia. Ecco un esempio: *inserito da tastiera un numero di giorni visualizzare a quanti secondi corrispondono.*

Il senso della freccia indica il flusso di esecuzione.



### Versione pseudo codice

```
leggi num_giorni
secondi_min ← 60
minuti_ora ← 60
ore_giorno ← 24
scrivi secondi_min*minuti_ora*ore_giorno*num_giorni
```

## **Selezione (decisione, scelta, alternativa)**

Il solo fatto di poter indicare un'istruzione dopo l'altra (sequenza) non ci consentirebbe di risolvere problemi interessanti. Ci sono innumerevoli situazioni in cui è necessario fare un 'controllo' (più tecnicamente 'verificare una condizione') ed agire di conseguenza **facendo eseguire blocchi diversi di istruzioni a seconda della situazione.**

**Versione pseudo codice:** scrivi (in rosso ed in corsivo la condizione che viene verificata)

**Se** *l'anno è bisestile*  
**allora**  
 considera febbraio con 29 giorni  
**altrimenti**  
 consideralo con 28.



**Se** *l'età è minore di 18* **allora** applica sconto **altrimenti** applica il prezzo pieno.



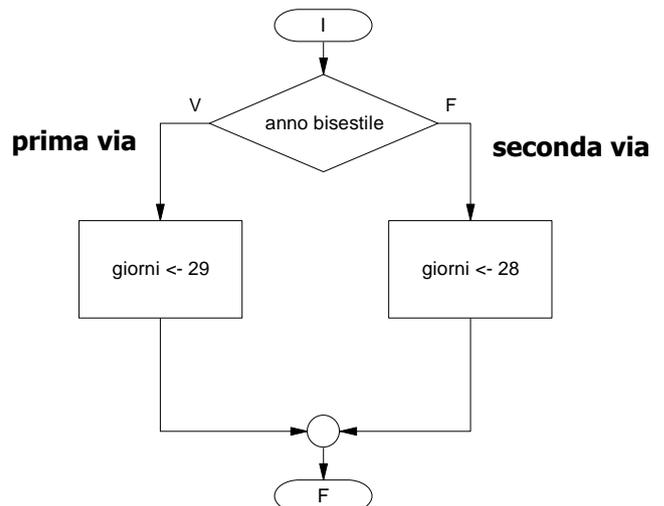
**Se** *la temperatura supera 37* **allora** fai suonare l'allarme.



Come avrete osservato, si controlla una **condizione** (anno bisestile ?, età minore di 18 ?, temperatura supera 37 ?) che può risultare *vera (true)* o *falsa (false)*. Può essere (primi due esempi) che vi siano operazioni da eseguire sia quando la condizione è vera sia quando è falsa (vedi sotto: selezione a due vie). E' però possibile (terzo esempio) che non ci sia nulla da fare quando la condizione è falsa (vedi sotto: selezione a una via).

Ecco come si rappresenta in un *flow chart* la struttura selettiva (a sinistra lo pseudo codice):

**Se** *l'anno è bisestile* **allora**  
 considera febbraio con 29 giorni  
**altrimenti**  
 consideralo con 28



Questo tipo di selezione è detto a 2 vie

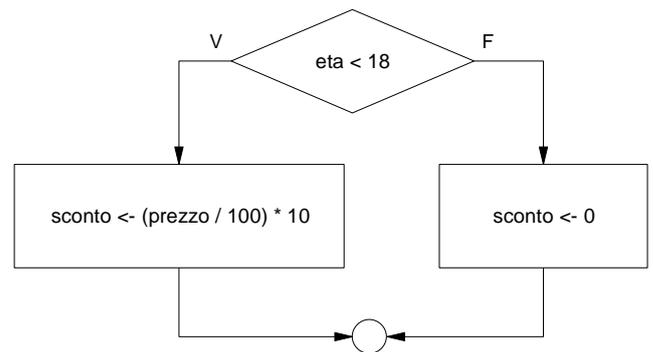
La condizione deve essere scritta all'interno del rombo. Il flusso di esecuzione si divide a seconda del risultato del controllo (**V sta per Vero e F sta per Falso**) e solo la strada 'a destra' o 'a sinistra' viene percorsa. Le istruzioni nella sezione 'vera' (o 'falsa') possono essere anche molte e non una sola come nel nostro esempio. Quando le istruzioni da eseguire a condizione vera (o falsa) sono terminate il flusso si ricongiunge usando il simbolo del cerchietto chiamato **connettore**: ○

Attenzione: nel rombo potete mettere solo condizioni, non assegnamenti!



Ecco un altro esempio (a sinistra lo pseudo codice):

Se l'età è minore di 18 allora  
 applica sconto  
 altrimenti  
 applica prezzo pieno.



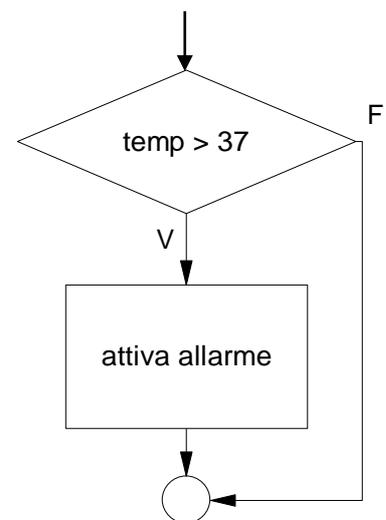
Nota: nel *flow chart* immaginiamo che le variabili *eta* e *prezzo* abbiano già ricevuto un valore valido per lo svolgimento dei calcoli. Lo sconto (10%) viene calcolato dividendo il prezzo pieno per 100 (calcolando così l'1%) e moltiplicando il risultato per 10 (ottenendo il 10%).

(a sinistra lo pseudo codice)

Se la temperatura supera 37 C allora  
 fai suonare l'allarme.

Questo tipo di selezione è detto ad una via

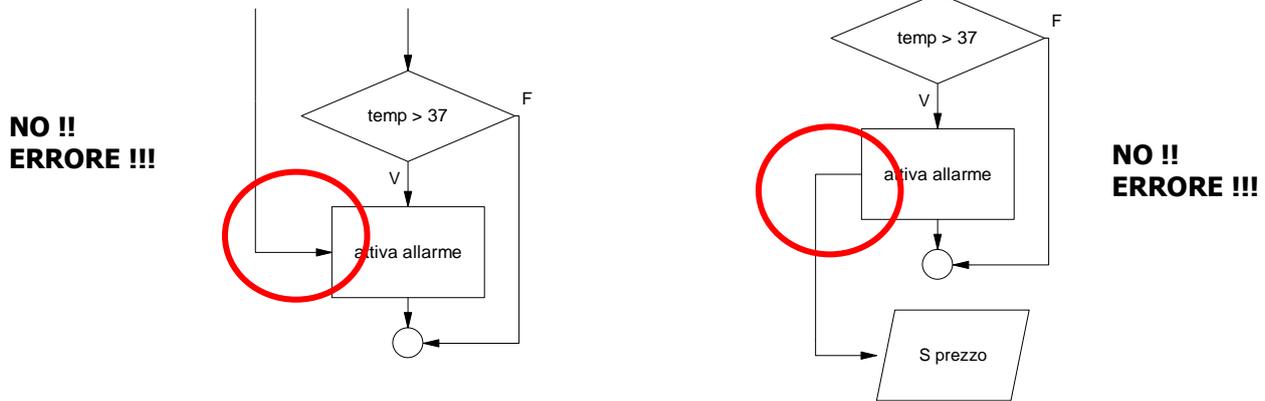
Come potete vedere se non c'è nulla che deve essere fatto quando la condizione è falsa, si congiunge la parte 'falso' direttamente con il connettore (il cerchietto).



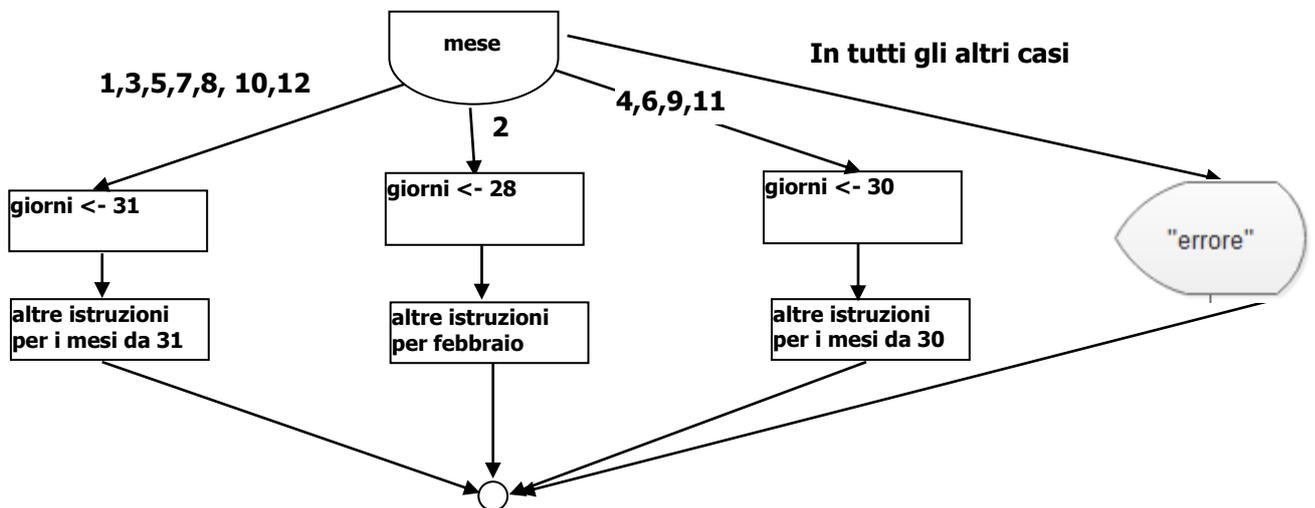
***Nel linguaggio C/C++ la struttura selettiva corrisponde all' if ... else.***

**NOTA IMPORTANTE:** in ogni diagramma che rappresenta la selezione ci deve essere sempre **un solo punto di ingresso** (la freccia entrante nel rombo) ed **un solo punto di uscita** (il connettore finale).

Evitate quindi frecce che dall'interno escono verso altri punti del diagramma o che dall'esterno giungono in un punto interno:



Esiste anche un'utile variante che prevede molte vie a seconda del valore di una variabile. Immaginiamo che la variabile *mese* contenga un valore da 1 a 12 che rappresenta uno dei mesi dell'anno. Di nuovo, vorremmo sapere quanti giorni considerare dato un certo mese. Vi ricordo che i mesi con 31 giorni sono 1 (gennaio), 3 (marzo), 5 (ecc.), 7, 8, 10, 12; quelli con 30 giorni sono 4 (aprile), 6, 9, 11; immaginiamo per semplicità che febbraio (2) ne abbia sempre 28.



Il flusso può seguire in questo caso tre vie (e non ci sarebbero problemi a metterne di più) a seconda del valore della variabile *mese*. I flussi possibili si ricongiungono poi nel solito connettore. E' anche previsto un flusso da seguire quando non si verifica nessuno dei casi precedenti (freccia 'in tutti gli altri casi').

### *Versione pseudo codice*

Se mese vale 1,3,5,7,8,10,12:

giorni ← 31

... altre istruzioni

se vale 2:

giorni ← 28

... altre istruzioni

Se vale 4,6,9,11:

giorni ← 30

... altre istruzioni

per tutti gli altri valori:

errore

**Nel linguaggio C/C++ la struttura di selezione  
a molte vie corrisponde allo switch**

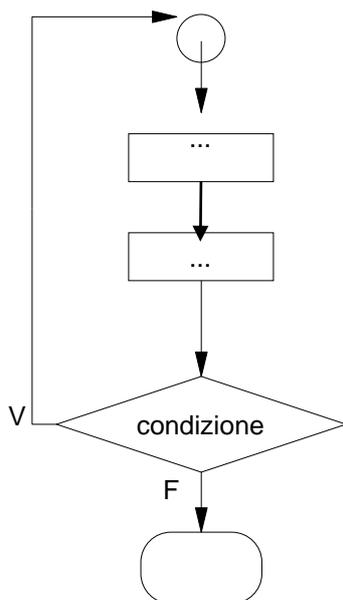
### Iterazione (ripetizione, cicli)



Pur avendo a disposizione sequenza e selezione rimangono molte le situazioni 'intrattabili' usando solo queste strutture. Consideriamo infatti questo problema all'apparenza molto semplice: stampare i numeri da 1 a 10000. Certamente la soluzione di usare per 10000 volte l'istruzione di scrittura sul video per stampare ogni numero non è molto praticabile ...

A parte l'enorme dimensione del *flow chart* (e del programma conseguente) ed il tempo necessario al disegno saremmo costretti a modificare lo schema, anche pesantemente, al variare della richiesta (stampare solo i primi 5000 numeri o i numeri fino al 15000).

Partendo invece dalla considerazione che si stanno ripetendo istruzioni molto simili (cambia solo il valore da scrivere sul video) si perviene ad una **struttura ciclica** che fa ripetere una od un gruppo di istruzioni fintanto che una certa condizione rimane vera.



#### Ecco lo schema generico di un *flow chart* per la struttura iterativa.

Le istruzioni (rappresentate dai due rettangoli interni) sono ripetute una prima volta. Giunti alla condizione, se questa è ancora vera (V) allora si torna all'inizio del ciclo e si ripete l'esecuzione dello stesso blocco di istruzioni. Quando la condizione è falsa (F) il ciclo termina e si prosegue.

Se all'interno del ciclo una delle istruzioni farà diventare falsa la condizione il ciclo terminerà. Diversamente procederà all'infinito (come nel controllo di un processo industriale che potrebbe non doversi fermare 'mai').

Questo tipo di ciclo è detto con **ripetizione per vero** (o a **uscita per falso**) e con **controllo in coda** (cioè alla fine del ciclo) o con **post-condizione (postcondizionale)**.

#### Versione pseudo codice

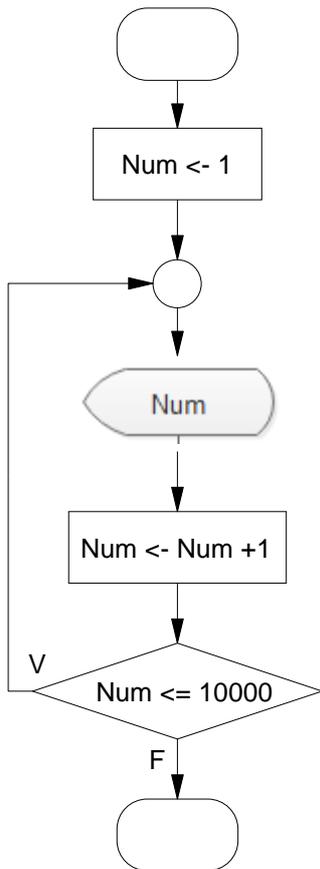
```

ripeti
  istruzione A
  istruzione B
  ... altre ...
finchè condizione è vera
    
```

NOTA: con questo tipo di ciclo le istruzioni vengono eseguite almeno una volta.

**In C++ questo tipo di iterazione corrisponde al ciclo `do ... while`**

Vediamo il *flow chart* completo per la stampa dei numeri da 1 a 10000:



*Num* è usato per controllare l'uscita dal ciclo. Esso viene impostato al valore 1 *prima* di iniziare il ciclo.

Il ciclo inizia stampando sul video il valore di *Num* (la prima volta stamperà quindi 1).

*Num* viene quindi incrementato di 1.

Se *Num* è ancora minore o uguale a 10000 si ripete il ciclo altrimenti il ciclo termina.

***Versione pseudo codice***

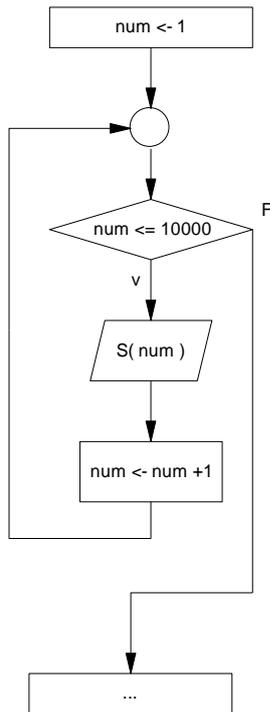
Num <- 1

ripeti

  scrivi Num

  Aggiungi 1 a Num

finchè Num<10000



Ecco invece un esempio di *flow chart* per l'altro tipo fondamentale di ciclo, quello detto come prima con **ripetizione per vero** o **uscita per falso** ma con **controllo in testa** (all'inizio) o con **pre-condizione (precondizionale)**.

Il ciclo inizia proprio con il controllo della condizione; se è vera si prosegue altrimenti il ciclo si interrompe.

NOTA: questo tipo di ciclo potrebbe anche non eseguire neanche una volta le istruzioni se la condizione fosse da subito falsa.

Gli ultimi due esempi costituiscono anche lo schema di un ciclo che deve essere ripetuto un certo numero di volte prefissato: la variabile *num* svolge il ruolo di *contatore* del numero di volte che è stato eseguito il ciclo. Ovviamente non saremmo di per sè costretti a stampare il valore di *num* ad ogni passaggio.

**Versione pseudo codice**

```

num <- 1
Finchè num<=10000
  scrivi num
  aggiungi 1 a num
  
```

**Nel linguaggio C/C++ questa struttura corrisponde al while**

SU [WWW.CAMUSO.IT](http://WWW.CAMUSO.IT) TROVERETE UN ESERCIZIARIO

## La sicurezza nei laboratori informatici e sul lavoro

Materiale tratto dal sito [www.geometri.arezzo](http://www.geometri.arezzo) e prodotto dall'I.T.E. "M. Buonarroti" di Arezzo, adattato ed integrato dal prof. Camuso.

Fino al XX secolo i lavoratori che si infortunavano rimanevano disoccupati e non ricevevano assistenza. Con l'introduzione delle prime norme, i lavoratori iniziarono a ricevere dei risarcimenti (inizia il grande sviluppo delle assicurazioni). Negli ultimi anni i legislatori hanno preferito puntare sulla prevenzione degli infortuni rispetto al risarcimento post-infortunio.

(Costituzione: Parte I - Diritti e doveri dei cittadini - Titolo III – Rapporti economici)

- ❖ Art. 41: II° comma Non può svolgersi in contrasto con l'utilità sociale o in modo da recare danno alla sicurezza, alla libertà, alla dignità umana.

Il riferimento legislativo attuale è il Testo Unico 81/08 anche se molti di voi avranno sentito parlare della famosa 'Legge 626' del 1994 che aveva profondamente rinnovato la legislazione precedente. Il Testo Unico si chiama così perché raccoglie in forma coerente ed organica tutte le precedenti leggi in materia di sicurezza sul lavoro.

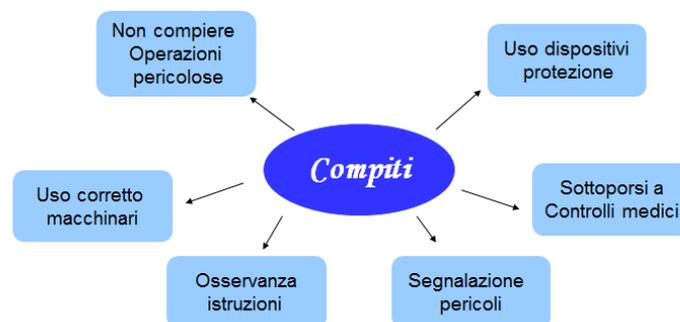
In particolare riguarda:

- Luoghi di lavoro;
- Uso delle attrezzature di lavoro e dei dispositivi di protezione individuale;
- Cantieri temporanei e mobili;
- Segnaletica su salute e sicurezza sul lavoro;
- Attrezzature munite di videoterminali;
- Protezione da atmosfere esplosive;
- Disposizioni diverse in materia penale e di procedura penale;

Gli allievi sono equiparati ai lavoratori:

- in relazione alla frequenza e all'uso dei laboratori appositamente attrezzati, dove gli allievi stessi possono essere esposti ad agenti chimici, fisici e biologici oppure utilizzino macchine, apparecchiature e strumenti di lavoro in genere, **compresi i computer**;
- nei periodi della settimana o della giornata in cui gli allievi sono effettivamente nei laboratori e utilizzano le attrezzature in essi contenute;
- se i programmi o le attività d'insegnamento (stabiliti anche a livello di singolo istituto e inseriti quindi nel P.O.F.) prevedono esplicitamente la frequenza e l'uso dei suddetti laboratori.

### Che compiti svolge il lavoratore?



## Rischio Elettrico e di incendio

Chiaramente alcuni suggerimenti hanno senso per la gestione delle apparecchiature a casa propria; è infatti compito dei tecnici scolastici, ad esempio, verificare che i cavi di alimentazione siano adatti nei laboratori (anche se è sempre meglio sapersi accorgere di evidenti violazioni o uso di cavi in modo che sembra non sicuro!)

- Verificare che i cavi di alimentazione siano idonei: un filo sottile un paio di millimetri non può sopportare 100 o 200 watt senza rischiare di fondere (rischio incendi); molto pericolosi anche i fili scoperti o con parti esposte in prossimità delle spine
- Fissare cavi di alimentazione: un cavo volante può causare inciampi e cadute, può causare stress alle apparecchiature (con continue accensioni/spegnimenti o fluttuazioni della tensione molte deleterie, ad esempio, per i microprocessori)
- Corretta strutturazione dei collegamenti elettrici: prese e spine devono essere dotati della 'presa a terra' ed avere quindi tre pioli, non due (eccetto le 'tedesche' che integrano la presa a terra in altro modo), il sistema elettrico deve essere dotato del cosiddetto 'salva vita', gli impianti devono essere certificati da un installatore professionista; i laboratori devono essere dotati del quadro elettrico in grado di erogare il sufficiente quantitativo di corrente (i watt: la somma della potenza assorbita da tutti i dispositivi collegati al quadro non deve superare la sua capacità); a casa pensarci due volte prima di dotare il pc di un mega alimentatore di 1200 watt se non potete collegarlo ad una presa adeguata. Tenete conto che la potenza erogabile da un intero appartamento è di 3000 watt; 1200 vorrebbe dire usarne la metà solo per il pc (per essere corretti assorbirebbe i 1200 watt solo se stipato di schede interne voraci come schede video al top e parecchi hard disk/lettori ottici).



- LE PRESE E SPINE ELETTRICHE DEVONO ESSERE IN BUONO STATO.

- NON DEVONO ESSERCI SOVRACCARICHI SULLE SINGOLE PRESE. QUESTI POSSONO CAUSARE INCENDI;



- Tutti i dispositivi devono essere lontani da calore, umidità, acqua. Ecco perché nei laboratori i pc devono essere ad una certa distanza dai caloriferi e devono nei limiti dello spazio essere lontani tra loro; i cavi non dovrebbero essere a terra (a meno che siano protetti da canaline calpestabili) perché una perdita d'acqua da un calorifero o da un condizionatore potrebbe compromettere la loro sicurezza e la vostra. Ed ecco perché è vietato bere in prossimità dei pc: uno spintone o una perdita di equilibrio spontanea potrebbe causare il rovesciamento di liquido su tastiere, torrette di alimentazione ecc.
- Quando si scollega una apparecchiatura dall'impianto elettrico: prima la si spegne (attenzione: per i PC spesso c'è anche un interruttore ON/OFF sugli alimentatori); poi si stacca la spina dal muro / ciabatta e solo alla fine, se previsto, si stacca il cavo dall'apparecchiatura

**INCENDIO:** In caso l'incendio si sviluppi in un luogo con presenza di Videoterminali è opportuno usare l'estintore ad Anidride Carbonica (CO<sub>2</sub>). L'estintore va controllato periodicamente (ogni 6 mesi), va utilizzato una sola volta dopo di che va contattata l'assistenza.

## Ergonomia

L'ergonomia è la disciplina che studia le condizioni e l'ambiente di lavoro per adattare alle esigenze psicofisiche del lavoratore. Moltissime persone passano la maggior parte della loro giornata davanti al computer, tenendo posture scorrette o altre abitudini, che possono causare molti problemi di salute.

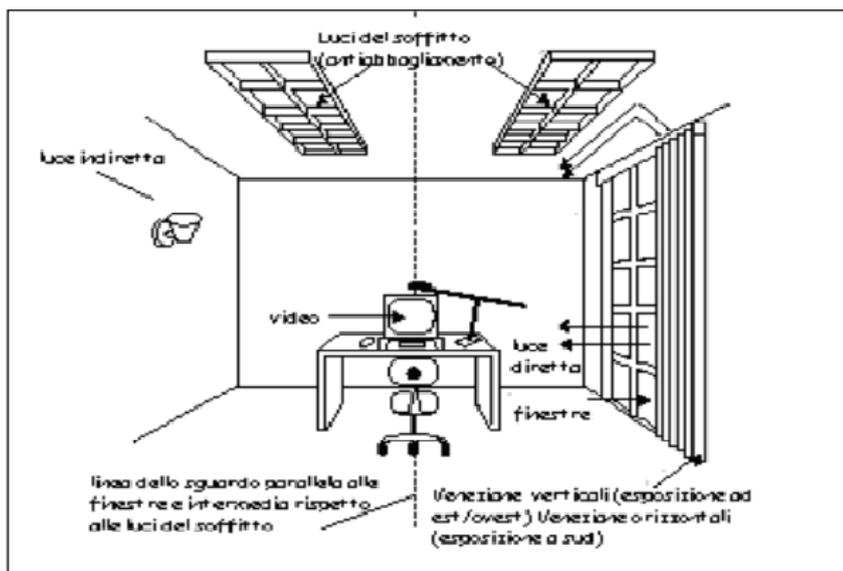
Le persone che lavorano per almeno 20 ore settimanali (videoterminalisti) al computer devono prendere, ogni due ore, una pausa di 15 minuti. In questo tempo l'operatore dovrebbe: cambiare attività, rilassare la vista guardando spazi lontani, compiere esercizi per distendere dita, cervicale e schiena.

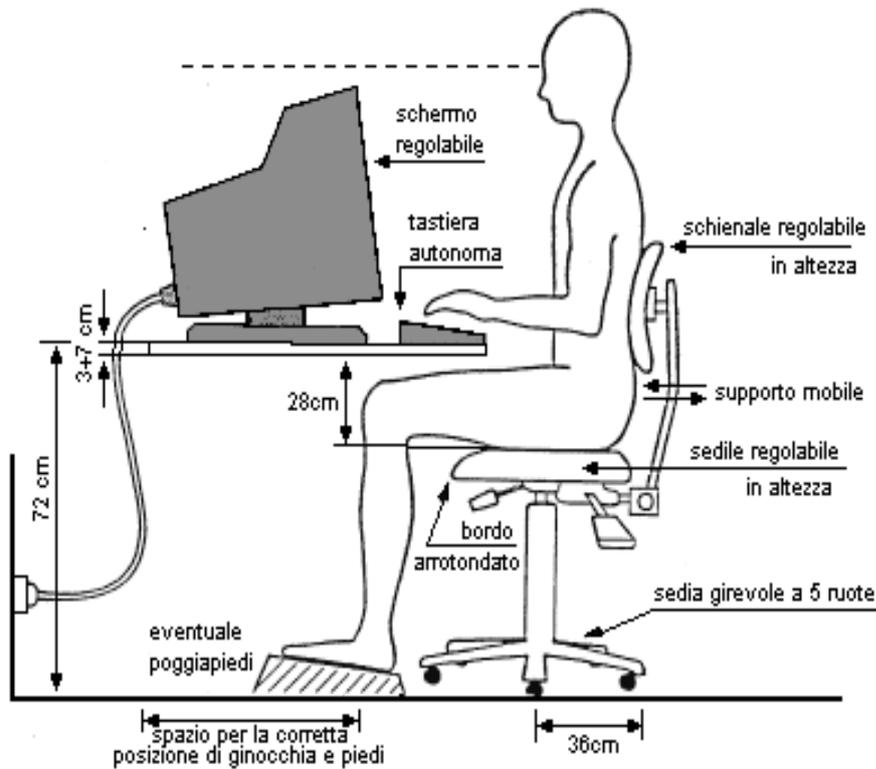
I fattori che possono causare problemi sono molteplici:

- L'illuminazione
- La sedia
- Il tavolo
- La tastiera
- Il mouse
- Il monitor
- La postura

Il monitor :

- deve essere circondato da superfici opache e non riflettenti
- deve essere perpendicolare alle finestre e dalla parte opposta ci dovrebbero essere luci artificiali





## LA POSTURA IDEALE



- Piedi appoggiati al pavimento

Per prevenire disturbi muscolo-scheletrici è importante assumere una posizione corretta di fronte al monitor:

- schiena dritta ben appoggiata allo schienale della sedia nel tratto lombare



- evitare posture fisse per tempi prolungati.

- Sedersi di fronte al monitor in modo da averlo perfettamente perpendicolare alla propria linea visiva.
- Impostare lo schermo a parametri di colore, luminosità e contrasto in modo da renderli il più confortevoli possibili.
- Ogni 20 - 30 minuti è bene distogliere la vista dallo schermo e spostare la propria attenzione su un oggetto posto a lunga distanza. Questo permette di cambiare la messa a fuoco dei nostri occhi per defaticarli.