



# APPUNTI di PROGRAMMAZIONE STRUTTURATA

*Rev. 1.2.3 b*





## INDICE GENERALE

### CAPITOLO 1

- 1.0 LA PROGRAMMAZIONE STRUTTURATA
- 1.1 TEOREMA DI JACOPINI-BÖHM
- 1.2 CHE COS'E' UN BLOCCO
- 1.3 LE STRUTTURE FONDAMENTALI
  - 1.3.1 DEFINIZIONE DI STRUTTURA SEQUENZIALE
  - 1.3.2 DEFINIZIONE DI STRUTTURA CONDIZIONALE ( O DI SELIZIONE)
  - 1.3.3 DEFINIZIONE DI STRUTTURA ITERATIVA

### CAPITOLO 2

- 2.0 CONCETTO DI ALGORITMO
- 2.1 FLOW CHART
- 2.2 ESEMPIO: REALIZZAZIONE DI FLOW CHART PER UN CICLO "FOR"
- 2.3 CONCETTO DI ANALISI
  - 2.3.1 ANALISI: TOP DOWN
  - 2.3.2 ANALISI: BUTTOM UP
  - 2.3.3 PROCESSO DI ASTRAZIONE
  - 2.3.4 ANALISI DI SEMPLICI PROBLEMI
- 2.4 ESERCIZI DI ANALISI E REALIZZAZIONE DI FLOW CHART



## CAPITOLO 3

- 3.0 PSEUDO-CODICE E PSEUDO-LINGUAGGIO
- 3.1 CONCETTO DI TIPO E DI DICHIARAZIONE DI DATO
- 3.2 DISTINZIONE TRA TIPI DI DATI PRIMITIVI E STRUTTURATI
- 3.3 GLI OPERATORI
- 3.4 TIPI DI DATI PRIMITIVI PREDEFINITI
  - 3.4.1 TIPO INTEGER O INT
  - 3.4.2 TIPO REAL O FLOAT
  - 3.4.3 TIPO BOOLEAN O BOOL
- 3.5 STRUTTURE DI DATI
  - 3.5.1 L' ARRAY
  - 3.5.2 IL RECORD
  - 3.5.3 L'INSIEME
- 3.6 RAPPRESENTAZIONE DELLE STRUTTURE DI DATI DI TIPO: ARRAY, RECORD ED INSIEME
  - 3.6.1 RAPPRESENTAZIONE DI UN ARRAY
  - 3.6.2 RAPPRESENTAZIONE DEL RECORD
  - 3.6.3 RAPPRESENTAZIONE DELL'INSIEME
- 3.7 INTRODUZIONE ALLE STRUTTURE DI DATI DINAMICHE
  - 3.7.1 FILE SEQUENZIALI
  - 3.7.2 PUNTATORI
  - 3.7.3 LISTE O CODE
  - 3.7.4 ALBERI BINARI
  - 3.7.5 ALBERI ORDINATI
  - 3.7.6 GRAFI
- 3.8 ESERCIZI

## CAPITOLO 4

- 4.1 MODULARITA' ED ASTRAZIONE
  - 4.1.1 CONCETTI DI PROGRAMMAZIONE MODULARE
  - 4.1.2 CONCETTI DI PROGRAMMAZIONE PER ASTRAZIONI
- 4.2 COSTRUTTI LINGUISTICI
- 4.3 DEFINIZIONI DI FUNZIONI
  - 4.3.1 METODOLOGIE DI PASSAGGIO DEGLI ARGOMENTI: PER VALORI E PER REFERENZA



## APPENDICI

- A. GERARCHIA DI ASTRAZIONI IN UN ELABORATORE
- B. CODICE ASCII
- C. PROTOTIPO O TIPO DEL VERO PROGRAMMATORE

## GLOSSARIO

## RINGRAZIAMENTI

## BIBLIOGRAFIA



## **NOTA DELL' AUTORE**

*Questa documentazione vuole essere una semplice guida per chi si inoltra, per la prima volta, nel mondo della programmazione strutturata.*

*Questa documentazione è stata creata prendendo il meglio degli argomenti contenuti in vari libri di testo universitari scritti in varie lingue. L'opera dell'autore sta nell'aver tradotto alcuni paragrafi e nell'aver cercato di armonizzare tra loro questi argomenti utilizzando un linguaggio alla portata di tutti (la dove è stato possibile), senza alterarne l'importanza ed il loro significato.*

*Con la presente nota si vuole mettere in evidenza il fatto che gli argomenti trattati non sono esposti nella loro completezza didattica, in quanto, per la totale comprensione, si richiederebbe un livello formativo molto tecnico<sup>1</sup>.*

*Questa documentazione non vuole essere considerata come una documentazione esaustiva. Si rimanda alla nota bibliografica, per chi volesse saperne di più.*

Ing. Giovanni Iozzelli

---

<sup>1</sup> Questa documentazione sarà soggetta a future revisioni.



## DITEMI QUELLO CHE PENSATE

Come lettori di questa documentazione, siete i più importanti critici e commentatori. Inviatemi la vostra opinione attraverso una e-mail<sup>2</sup>. Inoltre, mi piacerebbe conoscere se la documentazione vi è utile così com'è o cosa dovrei aggiungere per migliorarla. Come autore di questa dispensa vi assicuro che ogni sorta di commento è ben gradito. Mi scuso anticipatamente se non sarò in grado di rispondere a tutti.

Grazie

---

<sup>2</sup> Per inviare una e-mail scrivete a: [g.iozzelli@tidestudio.com](mailto:g.iozzelli@tidestudio.com)



## 1. INTRODUZIONE

Per programmazione strutturata si intende quella tecnica di programmazione all'interno della quale un lavoro viene suddiviso in vari passi (tasks), in cui ognuno di essi viene svolto da una unità indipendente di programma.

La programmazione strutturata semplifica il lavoro di programmazione in quanto prevede di frammentare un lavoro complesso in tanti piccoli “blocchi” ognuno dei quali svolge una elaborazione semplice e completamente indipendente dagli altri blocchi di programma, in quanto le variabili usate all'interno sono indipendenti da quelle usate al suo esterno.

Ognuno di tali blocchi potrà poi essere suddiviso in altri passi ancora più elementari fino alla soluzione del problema.

La programmazione strutturata semplifica anche il processo di “debugging” in quanto consente di verificare l'operato di ogni singolo “blocco” separatamente da tutti gli altri. Questo modo di operare comporta una “modularità” in quanto vi è la possibilità di riutilizzare all'interno di altri programmi un “blocco” già scritto per un altro programma.

L'utilizzo di questa metodologia di programmazione richiede l'ausilio di opportuni metodi di suddivisione del problema principale in più elementari sotto-problemi.

Tali sotto-problemi saranno tradotti in codici (utilizzando linguaggi di programmazione), per diventare “blocchi” (o moduli) risolutivi del complesso problema iniziale.



# CAPITOLO 1

## INDICE

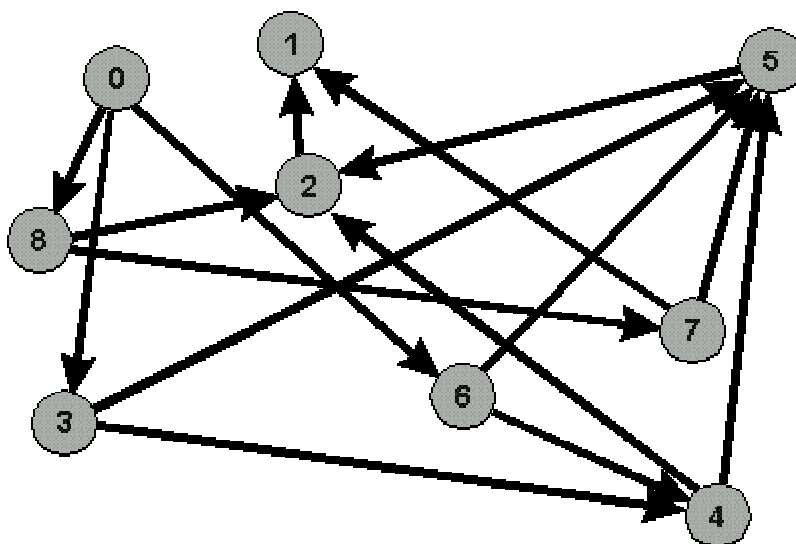
- 1.0 LA PROGRAMMAZIONE STRUTTURATA
- 1.1 TEOREMA DI JACOPINI-BÖHM
- 1.2 CHE COS'E' UN BLOCCO
- 1.3 LE STRUTTURE FONDAMENTALI
  - 1.3.1 DEFINIZIONE DI STRUTTURA SEQUENZIALE
  - 1.3.2 DEFINIZIONE DI STRUTTURA CONDIZIONALE ( O DI SELIZIONE)
  - 1.3.3 DEFINIZIONE DI STRUTTURA ITERATIVA





## 1.0 LA PROGRAMMAZIONE STRUTTURATA

Quando l'arte della programmazione muoveva i primi passi i programmatori consideravano dimostrazione di bravura programmare con il minor numero di istruzioni possibili, e questo faceva sì che si tendesse a riutilizzare pezzi di codice semplicemente saltando al loro inizio. Ne risultavano complicati collegamenti tra le varie parti del programma, non facilmente intelleggibili a chi non lo avesse scritto.



*L'aspetto grafico della computazione sarà il cosiddetto "piatto di spaghetti", con un groviglio di linee tra un punto e l'altro*

Il problema è che così facendo il programma risultava difficilmente comprensibile anche allo stesso programmatore, specie quando era grande o quando questo gli poneva mano dopo una lunga pausa. Inoltre, pur se questo modo di lavorare è ancora praticabile da un singolo programmatore, diventa del tutto inconcepibile per un lavoro di gruppo, quando è necessario che il codice sia facilmente capito anche da persone che non lo hanno scritto.



Si è arrivati dunque, verso la fine degli anni sessanta, a cambiare completamente la visione di cosa sia un programma ben scritto, ed a non considerare più la capacità di districarsi in mezzo ad aggrovigliate matasse di *goto* come principale caratteristica del buon programmatore.

Lo schema che alla fine è emerso, detto "*programmazione strutturata*", **si propone di dare l'aspetto di un flusso ordinato, tra un inizio ed una fine, a programmi che di per sé risultano intricati.**

Il modello da seguire è il "*programma sequenziale*", nel quale si applicano le varie operazioni una di seguito all'altra.

Però la semplice *sequenza* non può bastare per descrivere tutti i possibili problemi, che deve affrontare il programmatore nel tradurre la realtà in linee di codici, poiché non ha strumenti per tradurre la possibilità di scegliere tra due alternative.

Come si possono dunque conciliare queste due esigenze?

La risposta si trova osservando i processi naturali . Basta dunque imitare il linguaggio naturale.

Il *goto* è concepibile solo per una macchina i cui "pensieri" hanno dei ben determinati indirizzi, mentre in un linguaggio naturale un problema è già espresso in una forma strutturata che corrisponde allo "svolgimento temporale" di una particolare computazione, quello cioè che si chiama un "*processo*".

Nel linguaggio naturale sono presenti strutture operative del tipo "*fai questo ... , dopo fai quello ...* ", "*se ... fai questo ... altrimenti fai quello ...* ", "*finché ... fai così ...* " oppure "*ripeti 5 volte questo ...* ".

La tendenza ad imitare il linguaggio naturale è iniziata con Algol (1960), linguaggio che era appunto noto per la sua verbosità e che ha dato in eredità le sue strutture di controllo a quasi tutti i suoi successori.



L'utilizzo delle forme di controllo di flusso "alla Algol" (tecnicamente indicato con il termine: algoritmo) e cioè:

- **la sequenzialità**, nella quale un evento segue un altro;
- **l'alternativa**, nella quale il flusso si divide in due rami per poi riunirsi;
- **il ciclo**, in cui il flusso gira in tondo come in un gorgo

ponevano il problema se tutti i programmi potessero essere scritti solo con queste strutture, evitando l'ormai famigerato *goto*.

La risposta potrebbe essere evidente per un programma pensato fin dall'inizio in termini naturali, ma lo è molto meno se si richiede di riscrivere in forma strutturata un programma pieno di salti tra un punto e l'altro.

La cosa fu comunque risolta teoricamente nel 1966 **dal teorema di Jacopini-Böhm**.



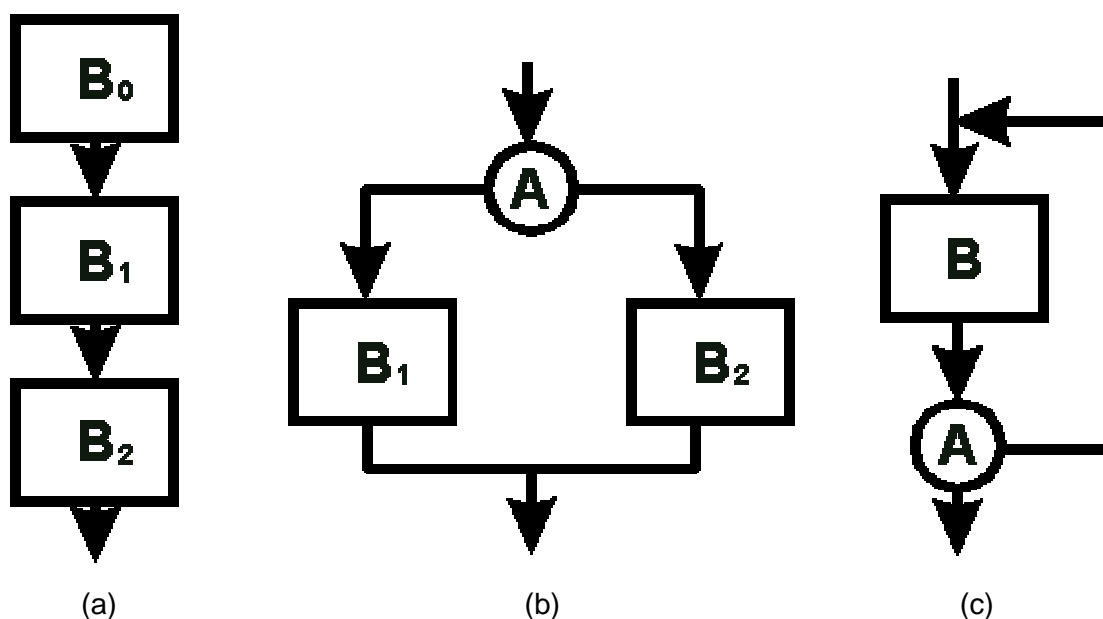
## 1.1 TEOREMA DI JACOPINI-BÖHM

Qualunque algoritmo può essere descritto utilizzando esclusivamente tre strutture fondamentali dette strutture di controllo:

- Struttura sequenziale
- Struttura condizionale (o di selezione)
- Struttura iterativa

*Definizione:* La programmazione che utilizza le suddette strutture, al fine di semplificare un algoritmo, è detta programmazione strutturata.

*Corollario:* Ogni programma può essere espresso con sequenze, con alternative o con cicli di blocchi di istruzioni.



Rappresentazioni grafica di struttura: (a) Sequenziale, (b) Condizionale, (c) Iterativa

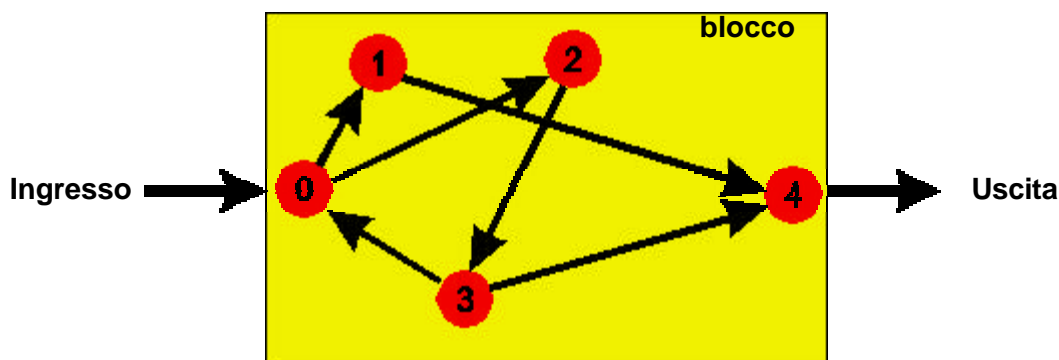
Prima di descrivere la logica di ognuna di queste strutture, dobbiamo prima definire la nozione di “blocco” all’interno di un programma.



## 1.2 CHE COS'E' UN BLOCCO

Il blocco di istruzioni è un sottoprogramma, cioè un insieme di istruzioni in sé completo, che non ha bisogno di far riferimento ad altro. E' importante notare che questo implica che il blocco ha un'unica uscita, la fine del blocco, e tutti i processi che eseguono istruzioni del blocco terminano al suo interno. Se infatti ci fossero delle "uscite laterali" tramite delle istruzioni di salto, queste dovrebbero far riferimento ai nomi (*etichette*) dei punti di programma a cui saltare, per cui il sottoprogramma non sarebbe più autonomo, facendo riferimento a queste etichette.

Inoltre si vuole che il blocco sia strutturato come una "scatola nera" (*black box*), nel senso che si possa utilizzare la sua capacità di elaborazione complessiva senza però vederne e poterne usare le singole parti. Dall'esterno non si può quindi saltare ad una particolare sua istruzione che sia diversa dal semplice inizio del blocco. Queste due caratteristiche fanno sì che i processi esecutivi, attraversando un blocco, trovino un'unica entrata (l'inizio del blocco), un'unica uscita (la fine del blocco) e nessuna "uscita laterale".



Rappresentazione grafica di "Blocco"

Questa, anche se di solito poco sottolineata, è la *caratteristica principale della programmazione strutturata*, perché prima di parlare del controllo del flusso (canali che si dividono o uniscono) bisogna sottolineare le caratteristiche di "**impermeabilità**" del



**tubo che porta questo flusso**, che sono appunto una sola entrata, una sola uscita e niente perdite laterali.

Dal punto di vista simbolico le parentesi sono il modo migliore di descrivere i blocchi:

$$\{ \dots \}^3$$

L'inizio di un blocco corrisponde al simbolo "{", la fine al simbolo "}", e i blocchi possono essere annidati uno dentro l'altro come sono i livelli di parentesi in matematica. Un blocco può essere trattato (e pensato) a tutti gli effetti come una singola istruzione, e quindi la combinazione più semplice è la sequenza di blocchi:

$$\{ B_1 \} \quad \{ B_2 \} \quad \dots \quad \{ B_N \}$$


*Rappresentazione grafica di un blocco.*

<sup>3</sup> Il simbolo della parentesi: graffa aperta “{” corrisponde alla combinazione di tasti ALT+123  
graffa chiusa “}” corrisponde alla combinazione di tasti ALT+125



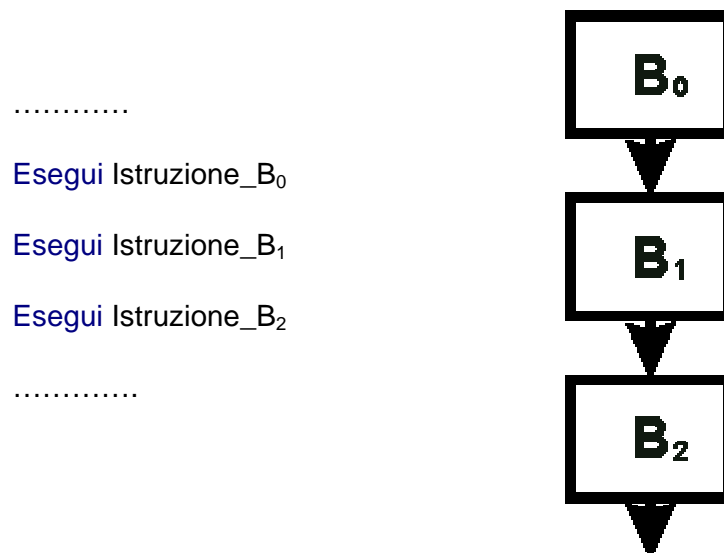
## 1.3 LE STRUTTURE FONDAMENTALI

### 1.3.1 DEFINIZIONE DI STRUTTURA SEQUENZIALE

E' la struttura che pone in sequenza azioni elementari (blocchi) *direttamente eseguibili una di seguito all'altra*.

$\{ B_0 \} ; \{ B_1 \} ; \{ B_2 \}$

Utilizzando un metodo di pseudo-codifica, il tutto si esprime come segue:



In tal modo è possibile costruire un flusso di operazioni che portano ad una certa informazione.



### 1.3.2 DEFINIZIONE DI STRUTTURA CONDIZIONALE ( O DI SELEZIONE)

E' la struttura che consente di eseguire una "scelta" tra due possibilità poste in alternativa, in base alla risposta di un test assegnato.

`if { A } then { B1 } else { B2 }`

Qui si ha la valutazione di una condizione "A": se questa è vera si esegue il primo blocco altrimenti si esegue il secondo <sup>4</sup>.

Utilizzando un metodo di pseudo-codifica, il tutto si esprime come segue:

.....

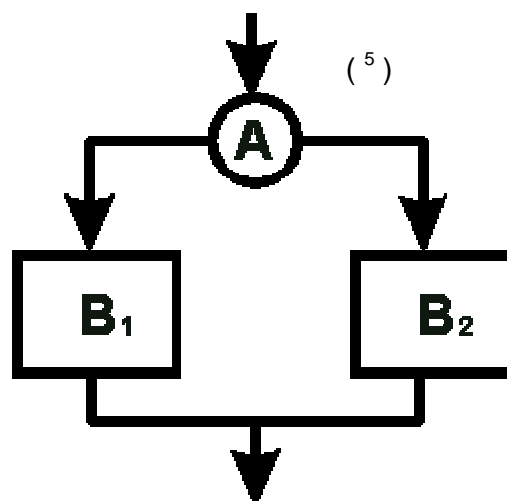
**SE** il Test A risulta VERO

**ALLORA** Esegui Istruzione B1

**ALTRIMENTI** Esegui Istruzione B2

**FINE SE**

.....



<sup>4</sup> I termini "then" ed "else" non sono necessari per la costruzione, servono solo per chiarezza linguistica.

<sup>5</sup> Qui adottiamo per le istruzioni decisionali dei cerchi invece dei soliti rombi. Questi ultimi sono usati perché sono sottointese due possibili risposte alternative, sì o no, corrispondenti a due vertici del rombo. Qui però pensiamo nel modo più generale, con scelta non tra due sole possibilità, ma tra quante si vuole (multibranch), per cui usiamo dei cerchi. Le corrette simbologie per creare i Flow Chart saranno spiegate più avanti.





Un' altra forma di struttura condizionale può essere "l'esecuzione opzionale".

`if { A } do { B }` oppure `if { not A } do { B }`

Utilizzando un metodo di pseudo-codifica, il tutto si esprime come segue:

.....

SE il Test A risulta VERO

Esegui Istruzione B

.....

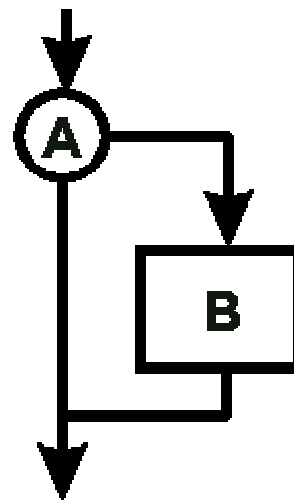
oppure

.....

SE il Test A risulta NON E' VERO

Esegui Istruzione B

.....





### 1.3.3 DEFINIZIONE DI STRUTTURA ITERATIVA

E' la struttura che consente di eseguire la ripetizione di una o più informazioni fino a quando la risposta di un test assegnato non è soddisfatta. Quando viene soddisfatta l'iterazione termina ( la definizione può essere letta anche in modo complementare).

La struttura iterativa è anche nota con il nome di “**ciclo**”. Il più facile da descrivere è quello che richiede di eseguire un certo numero *N di volte* un blocco di istruzioni. Esprimendoci in un pseudo-linguaggio di programmazione, possiamo scrivere:

For *i=1 to N* do { ..... }

Questo tipo di ciclo però è facilmente esprimibile usando quello generale, che valuta una condizione per l'esecuzione. Di questo tipo di ciclo ci sono due forme <sup>6</sup>:

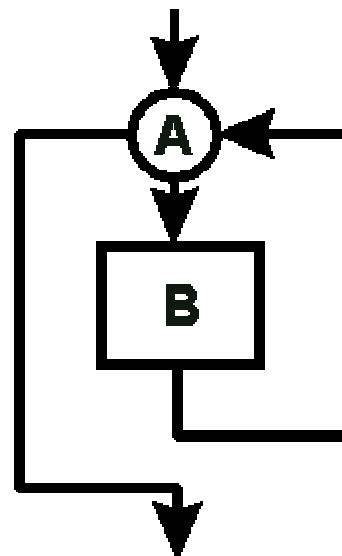
1) while (A) do { B }

.....

Sinchè A non è soddisfatto     // while A

Esegui il blocco B {...}        // do { B }

.....



Qui la condizione viene valutata all'inizio, quindi il blocco potrebbe anche non essere mai eseguito, se la condizione è subito falsa.

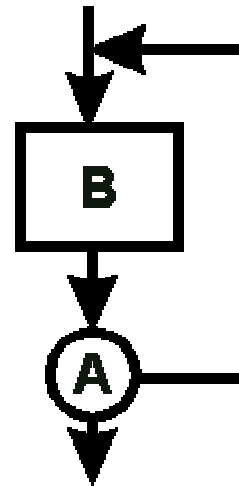
<sup>6</sup> Il costrutto dell'istruzione non vuole fare riferimento ad alcun specifico linguaggio di programmazione. Ogni corrispondenza a costrutti reali è da considerare una coincidenza.



oppure:

2) `do { B } while (A)`

.....  
Esegui il blocco B                    `// do { B }`  
Sinchè il test A è soddisfatto   `// while (A)`  
.....



Con questa struttura si esegue il blocco una prima volta, ed alla fine si valuta la condizione e, nel caso sia vera, si ripete finché la condizione non diventi falsa.

**Oltre a queste due forme si trovano costruzioni equivalenti che usano la condizione in modo opposto, cioè se è vera non eseguono il ciclo.**



## CAPITOLO 2

### INDICE

- 2.0 CONCETTO DI ALGORITMO
- 2.1 FLOW CHART
- 2.2 ESEMPIO: REALIZZAZIONE DI FLOW CHART PER UN CICLO "FOR"
- 2.3 CONCETTO DI ANALISI
  - 2.3.1 ANALISI: TOP DOWN
  - 2.3.2 ANALISI: BUTTOM UP
  - 2.3.3 PROCESSO DI ASTRAZIONE
  - 2.3.4 ANALISI DI SEMPLICI PROBLEMI
- 2.4 ESERCIZI DI ANALISI E REALIZZAZIONE DI FLOW CHART



## 2.0 CONCETTO DI ALGORITMO

Riassumendo quindi il contenuto del teorema di Jacopini-Böhm si può dire che: ogni programma può essere espresso organizzandolo in blocchi uno contenuto nell'altro, oppure in sequenza uno di seguito all'altro, con eventuali blocchi alternativi (if ... then ... else ...) o blocchi da ripetere in ciclo (while ... do ...).

Graficamente si hanno dei flussi che scendono dall'alto verso il basso, con i blocchi rappresentati da rettangoli e le istruzioni di decisione da cerchi. I blocchi sono organizzati in "piani", e le frecce passano da un piano a quello inferiore, suddividendosi all'uscita delle istruzioni di decisione. L'unica eccezione è che una freccia può risalire di un piano per rieseguire un blocco, nel qual caso si ha una ripetizione ciclica. All'interno di un blocco tutta questa struttura può ripetersi rispetto a dei sottoblocchi.

Questa struttura prende il nome di "**Algoritmo**" il cui andamento di flusso viene visualizzato utilizzando una rappresentazione grafica standard chiamata "**Flow chart**".



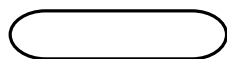
## 2.1 FLOW CHART

Il flow chart è un metodo di progettazione che modella il flusso del programma mediante una rappresentazione simbolica.

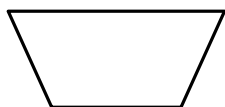
I flow chart permettono più livelli di dettaglio: è possibile quindi in prima misura fare uno schema ad alto livello delle macro-istruzioni che dovranno essere portate a termine, quindi successivamente dettagliate ogni singola macro-istruzione giù fino a raggiungere una mappatura 1:1 con le istruzioni software del linguaggio di programmazione scelto.

Qui di seguito sono rappresentati i simboli standard per la realizzazione di un flow chart. Questi sono stati suddivisi in due gruppi in base alla loro frequenza d'uso.

### 1° Gruppo Fondamentale di simboli



**INIZIO / PAUSA**



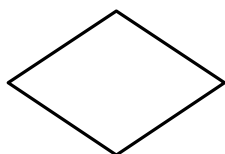
**LETTURA / SCRITTURA**



**CONNESSIONE / LINK**



**PROCESSO / ISTRUZIONE**



**DECISIONE**



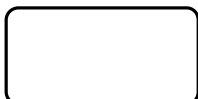
## 2° Gruppo di simboli



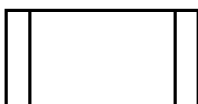
**INPUT MANUALE**



**DATI**



**ELABORAZIONE ALTERNATIVA**



**ELABORAZIONE PREDEFINITA**



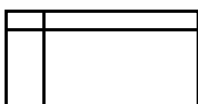
**DOCUMENTO**



**DOCUMENTO MULTIPLO**



**PREPARAZIONE**



**MEMORIA INTERNA**



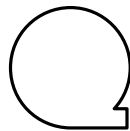
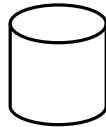
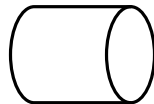
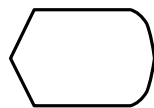
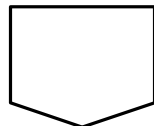
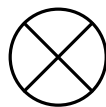
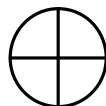
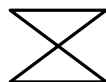
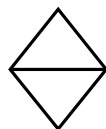
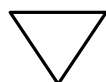
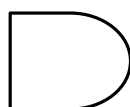
**DATI MEMORIZZATI**



**SCHEDA**



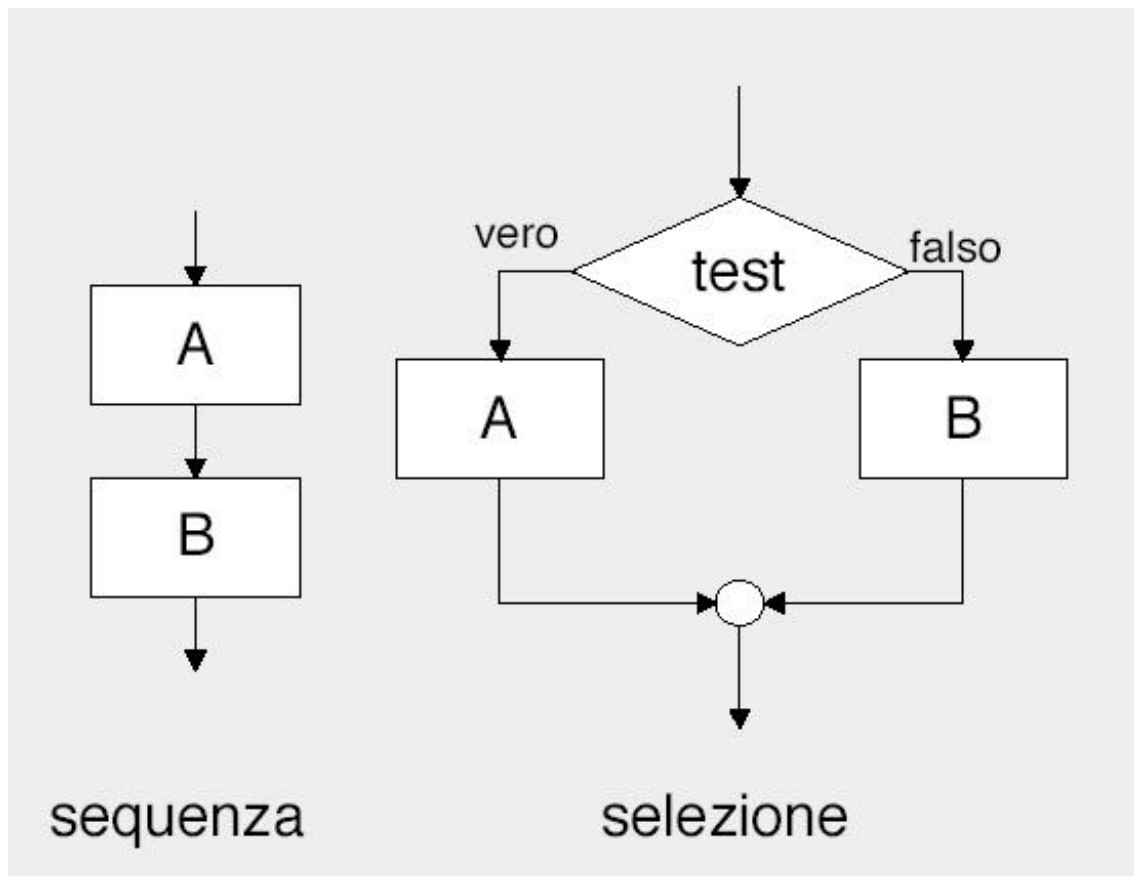
**NASTRO PERFORATO**

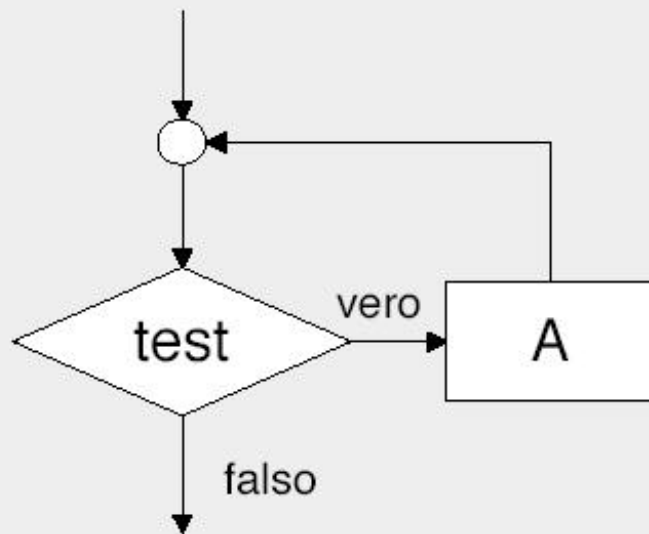
**MEMORIA AD ACCESSO CASUALE****DISCO MAGNETICO****MEMORIA AD ACCESSO DIRETTO****UNITA' DI VISUALIZZAZIONE GRAFICA****CONNETTORE PAGINA ESTERNA****SOMMA****O****FASCICOLAZIONE****ORDINAMENTO****ESTRAZIONE****FUSIONE****RITARDO**





Qui di seguito sono riportati i flow charts delle tre strutture fondamentali della programmazione strutturata, utilizzando gli appropriati simboli standardizzati:

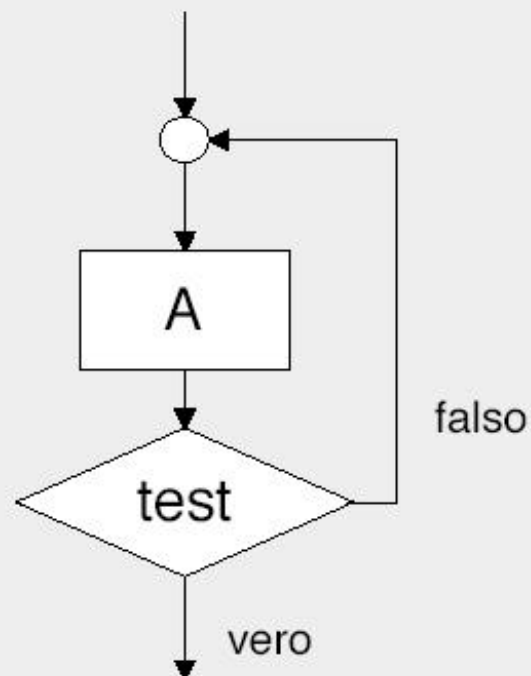




## iterazione

Il blocco "A" potrebbe non essere mai eseguito.

while (test) -> do [A]



## iterazione

Il blocco "A" viene eseguito almeno una volta.

repeat [A] until (test)



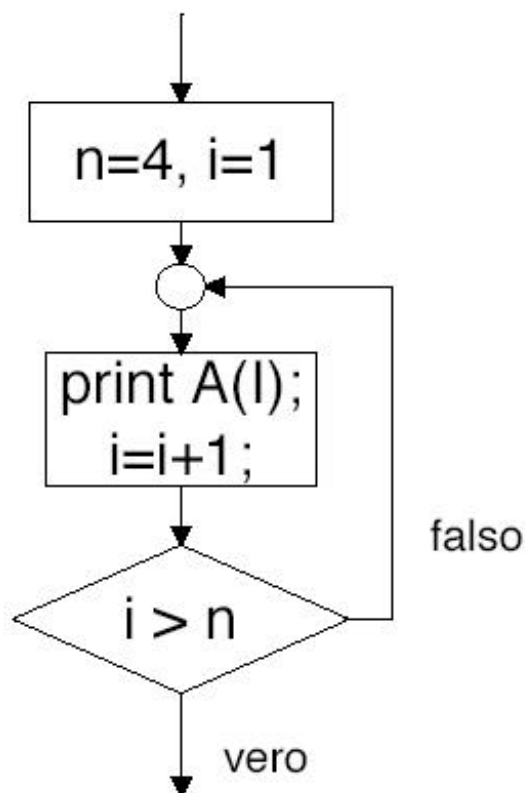
## 2.2 ESEMPIO: REALIZZAZIONE DI FLOW CHART PER UN CICLO “FOR”

*Problema:*

Realizzare il Flow Chart di un ciclo FOR che stampi per quattro volte al funzione A(I).

*Risoluzione:*

- Definire la variabile “i” che memorizza il numero di volte di esecuzione del ciclo;
- Definire una costante “n” ed assegnarli il valore 4;
- Definire il blocco di istruzioni che esegue la stampa della funzione e che incrementi il valore della variabile “i”
- Definire un test di controllo tra i valori di “i” e “n”



In pseudocodice:

```
... ..  
for i=1:4  
    Print A(I)  
end  
... ..
```

Esempio in C:

```
... ..  
for (i=1; i>n ; i++)  
    print A(I);  
... ..
```



### 2.3.0 CONCETTO DI ANALISI

La programmazione strutturata per essere applicata correttamente, necessita l'uso di apposite metodologie, senza le quali si rischia di perdere correttezza e modificabilità del programma.

Con la parola “metodo” si intende un insieme di regole generali e flessibili tendenti ad indirizzare le attività di “analisi” per il raggiungimento dell'obiettivo finale attraverso un processo di sviluppo a fasi successive.

I passi principali da seguire per realizzare un software “strutturato” sono:

- 1) L'analisi preventiva del problema:
  - Metodo Top-Down
  - Metodo Bottom-Up
- 2) Esecuzione di un “processo di astrazione” al fine di estrapolare solo le informazioni indispensabili per consentire l'elaborazione del problema da parte del computer (fase fortemente influenzata dalla tipologia di linguaggio che si utilizzerà per la realizzazione del progetto)
- 3) La suddivisione del problema in sotto-problemi più elementari (detti moduli) che svolgono compiti autonomi. Tali moduli devono essere il più possibile indipendenti nel senso di dipendere dagli altri moduli solo in termini di “dati” e non di “condizioni”. In questa fase si ripete l'analisi del problema utilizzando i metodo elencati al punto 1), ma limitata al sotto-problema.

Così si potrà ottenere un programma di alta qualità in termini di “correttezza” e “modificabilità”, suddiviso in moduli indipendenti che scambiano tra loro solo “dati”.



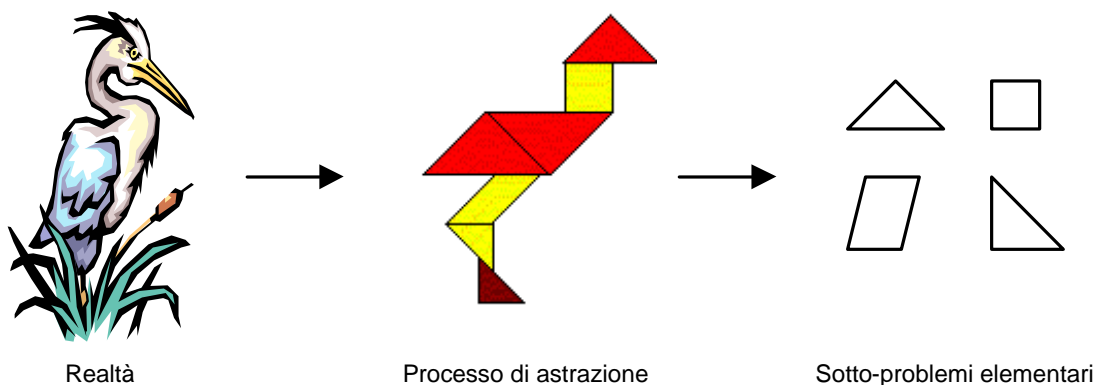
### 2.3.1 ANALISI: TOP-DOWN

Si esegue una procedura iterativa di frazionamento del problema in sottoproblemi più semplici, fino a che ogni sottoproblema può essere implementato facilmente nel linguaggio di programmazione scelto.

Caratteristiche di questo processo di analisi sono:

- La struttura del programma, o dell'algoritmo, risulta più chiara;
- Permette di affrontare il problema con un'opportuna angolazione: più ampia nelle fasi iniziali di analisi globale, più stretta (focalizzata) durante la definizione dei dettagli;
- Realizzazione di sottoprogrammi che possono essere riutilizzati in più parti del software di cui appartengono o in altri software;
- Se la scomposizione del problema è stata realizzata correttamente nei passaggi precedenti, i vari sottoproblemi possono essere risolti indipendentemente, e la loro struttura interna modificata, senza alterare la struttura globale del programma.

#### Processo di analisi di tipo TOP-DOWN



*Partendo da un modello reale si esegue un processo di astrazione per effettuare una semplificazione per poter estrapolare gli elementi caratteristici per poter realizzare il modello.*



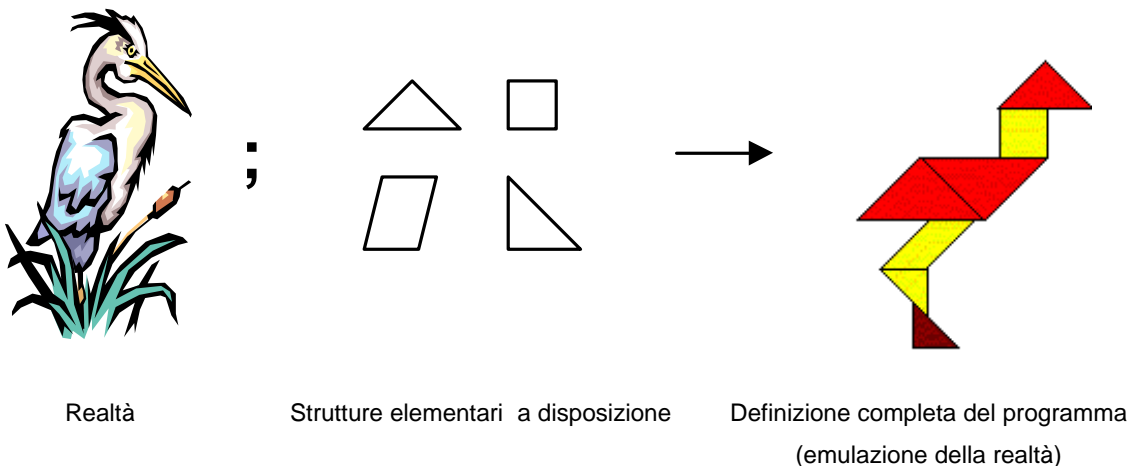
### 2.3.2 ANALISI: BUTTOM UP

Si parte dalla identificazione delle strutture o funzionalità elementari che vengono successivamente composte in frammenti più grossi, fino alla definizione completa del programma o dell'algoritmo.

Caratteristiche di questo processo di analisi sono:

- Processo utile quando si vuole costruire un programma utilizzando, dopo eventuali modifiche o integrazioni, programmi o sotto-programmi realizzati precedentemente;
- Se l'analisi del programma è complessa, questa può essere semplificata cercando di definire gli aspetti critici intorno ai quali viene realizzata il resto della struttura.

#### Processo di analisi di tipo BOTTOM-UP



*Partendo da un modello reale e confrontandolo con gli elementi in nostro possesso si costruisce il modello*



### 2.3.3 PROCESSO DI ASTRAZIONE

L'informazione che viene resa disponibile all'elaboratore consiste in un insieme selezionato di "dati" relativi al mondo reale, cioè un insieme che si ritiene sia rilevante per il problema, e dal quale si crede debbano derivare i risultati desiderati.

I dati rappresentano un'astrazione della realtà, nel senso che certe proprietà e caratteristiche degli oggetti reali vengono ignorate, perché sono marginali ed irrilevanti per il particolare problema in esame. Quindi un astrazione è anche una semplificazione dei fatti.

Per comprendere meglio questo concetto potremmo prendere ad esempio un archivio del personale di qualche azienda. In esso ogni impiegato è astrattamente rappresentato con un insieme di dati, che sono di interesse per il datore di lavoro o per l'amministrazione. L'insieme potrebbe contenere qualche identificazione dell'impiegato, per esempio, il suo nome, il suo cognome o il suo salario. Però, molto probabilmente, non conterrà informazioni irrilevanti come: il colore dei capelli, il peso o l'altezza.

Quando si risolve un problema, con o senza un elaboratore, è necessario scegliere un'astrazione della realtà, cioè definire un insieme di dati per rappresentare la situazione reale.

La scelta deve essere guidata dal problema che si vuole risolvere.

In seguito, occorre scegliere una rappresentazione dell'informazione. La scelta deve essere guidata dallo strumento che serve a risolvere il problema, cioè le caratteristiche dell'elaboratore. In molti casi, questi passi non sono completamente dipendenti.

La "scelta della rappresentazione" dei dati è spesso abbastanza difficile, e non è unicamente determinata dagli strumenti a disposizione. Essa deve essere sempre attuata tenendo conto di quali operazioni dovranno essere eseguite sui dati.

In questo contesto emerge l'importanza dei linguaggi di programmazione.

Un linguaggio di programmazione rappresenta un elaboratore astratto, capace di comprendere i termini del linguaggio, che potrebbero contenere un certo livello di astrazione degli oggetti utilizzati dalla macchina reale.



Usare un linguaggio di programmazione che offra una scelta conveniente di astrazioni, comuni alla maggior parte dei problemi di elaborazione dati, è importante principalmente in termini di affidabilità dei programmi che ne risultano.

E' più facile progettare un programma ragionando su nozioni familiari come numeri, insiemi, sequenze e ripetizioni, piuttosto che su bit, "parole" e salti. Naturalmente un elaboratore vero rappresenterà tutti i dati, siano essi numeri, insiemi o sequenze, con grandi raggruppamenti di bit.

Tutto ciò, però, diviene irrilevante per il programmatore, che non deve preoccuparsi dei dettagli di rappresentazione delle sue astrazioni, se è sicuro che le rappresentazioni scelte dall'elaboratore ( o compilatore) sia ragionevole per i suoi scopi.

Più le astrazioni sono vicine ad un dato elaboratore, più facile è, per l'ingegnere o per l'implementatore del linguaggio, prendere una decisione sulla rappresentazione, e maggiori sono le probabilità che una sola scelta si adatti a tutte ( o quasi) le applicazioni pensabili.

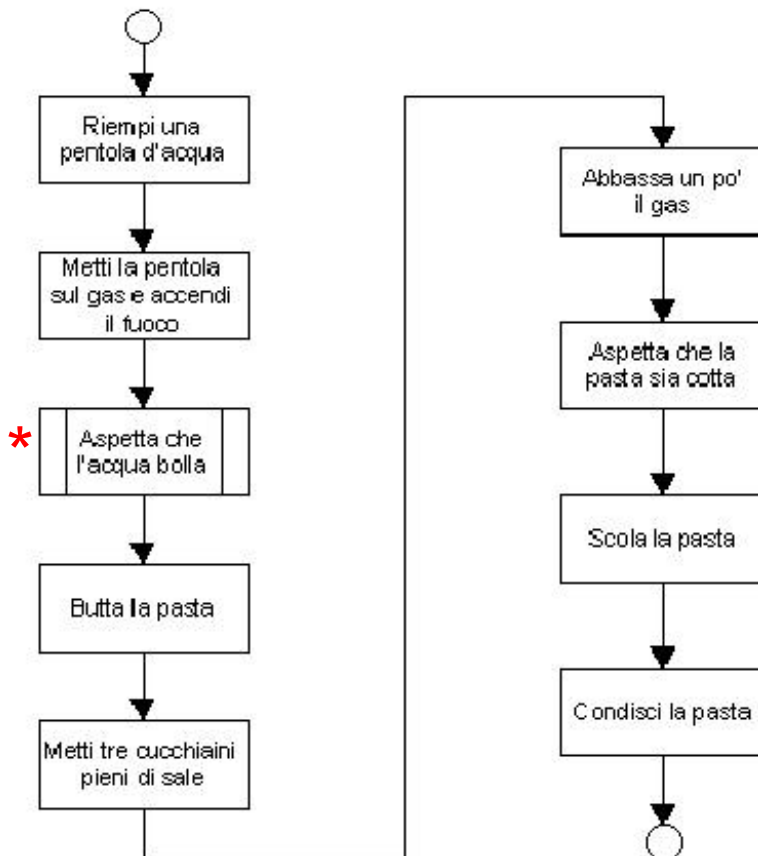




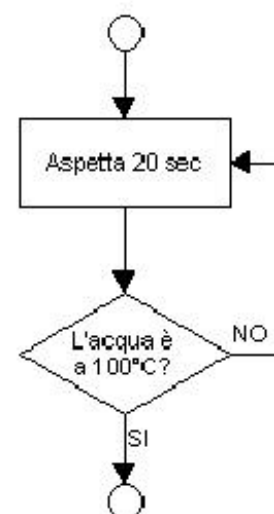
### 2.3.4 ANALISI DI SEMPLICI PROBLEMI

#### Problema 1:

Si disegni un flow chart che rappresenti il flusso di un programma per un robot che deve far cuocere la pasta asciutta, procedendo per macro-istruzioni



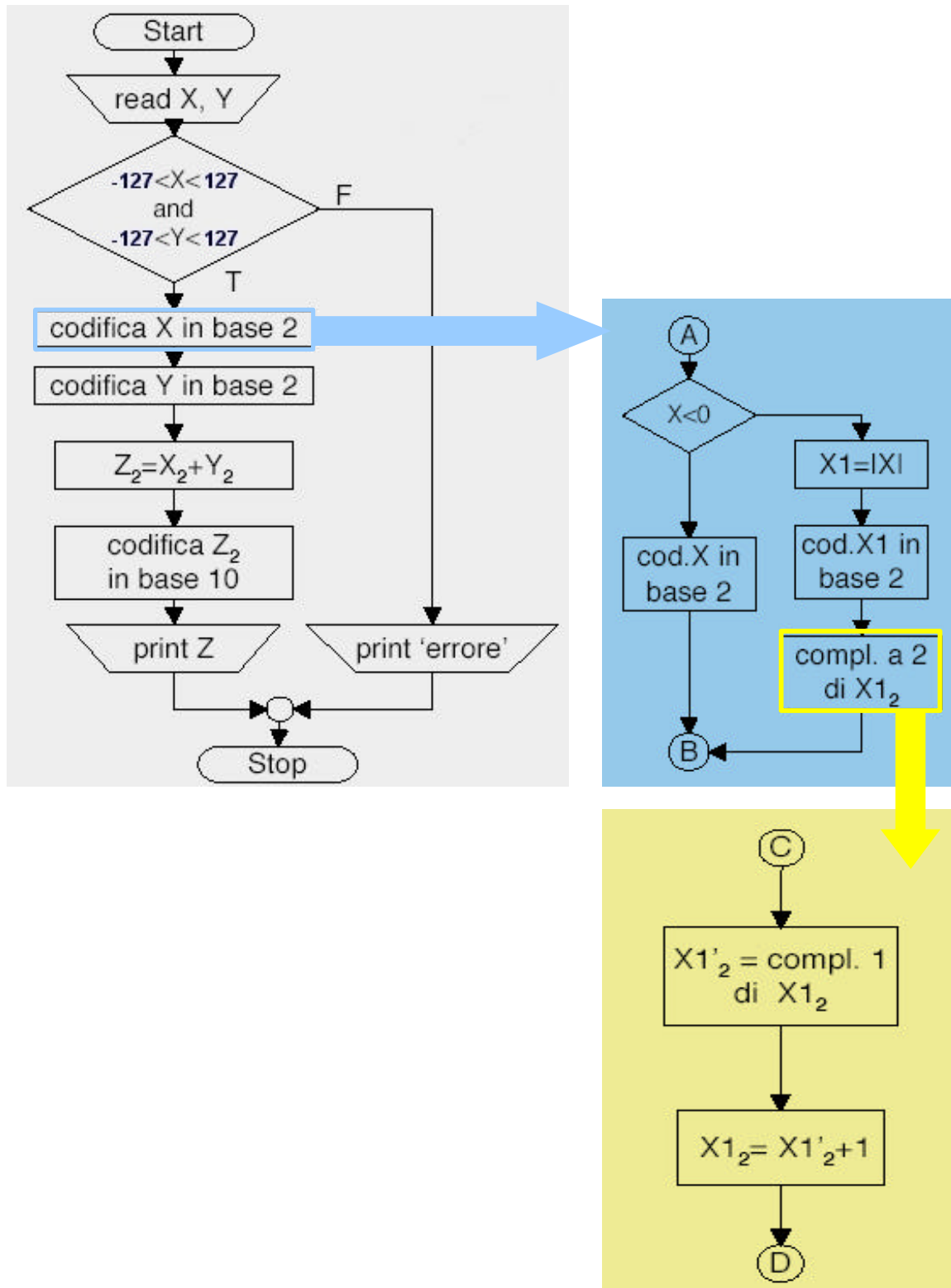
#### Procedura\* *Aspetta che l'acqua bolla*





*Problema 2 (utilizzo dell'analisi Top-Down):*

Si voglia rappresentare un flow chart per la realizzazione di un programma per l'addizione di due numeri X ed Y simulando il procedimento di addizione di due numeri binari a 8 bit.

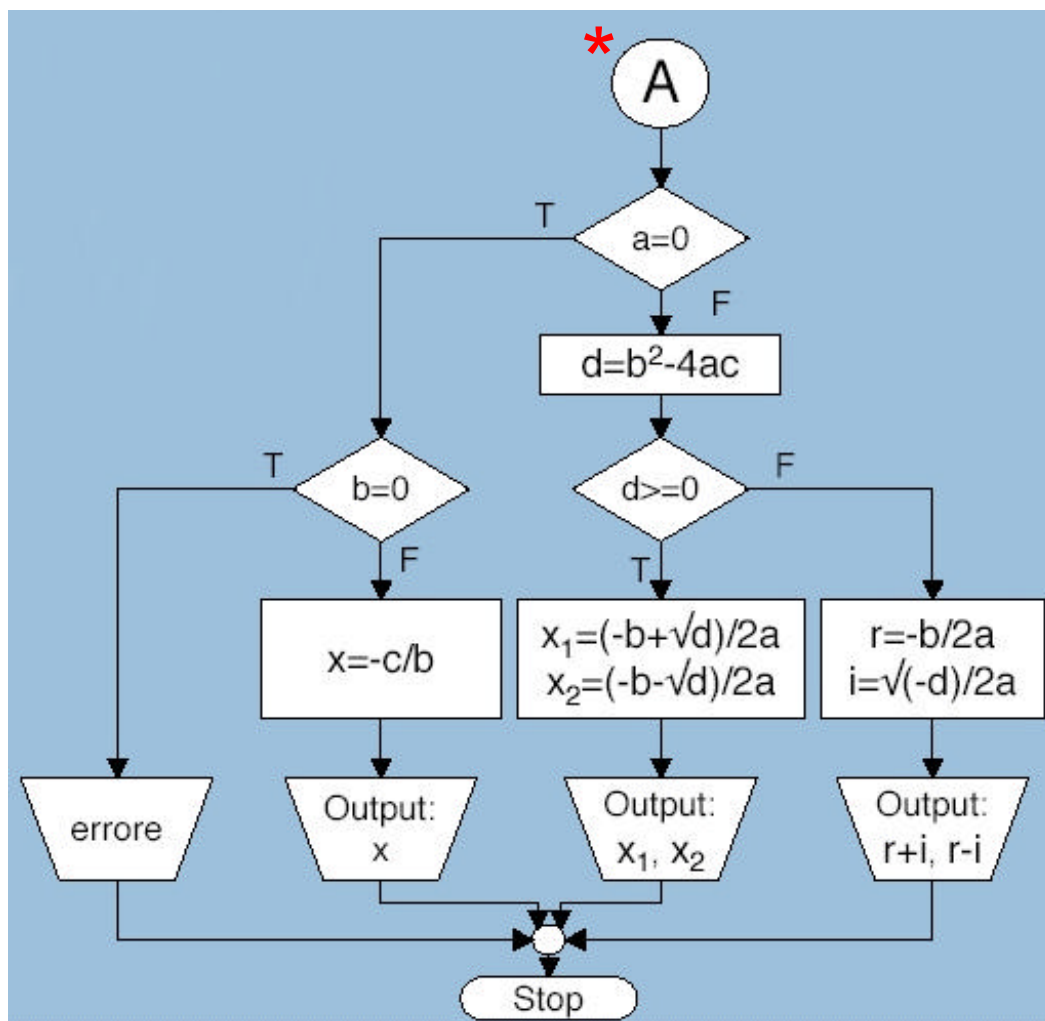
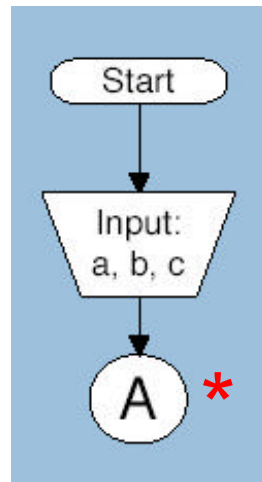


**Problema 3:**

Realizzare un algoritmo per la risoluzioni di equazioni algebriche di secondo grado del tipo:  $ax^2+bx+c=0$

T = True = Vero

F = False = Falso





## 2.4 ESERCIZI DI ANALISI E REALIZZAZIONE DI FLOW CHART

### *Esercizio 1*

Realizzare il diagramma di flusso con le operazioni da compiere per scrivere un file di testo utilizzando il NOTEPAD di Windows<sup>®</sup>, partendo dall'accensione del PC e contemplando eventuali errori (es: il programma non è installato, ecc.)

### *Esercizio 2*

Fare il diagramma di flusso con le operazioni da compiere per calcolare area e perimetro di un triangolo isoscele, supponendo che gli input siano:

- lunghezza della base e dell'altezza
- base e angolo (tra la base e un lato)

### *Esercizio 3*

Realizzare un diagramma di flusso che, date le dimensioni del pavimento in una stanza rettangolare, e dato il lato di una piastrella quadrata, trovi quante sono le piastrelle da utilizzare per la pavimentazione.

### *Esercizio 4*

Realizzare un diagramma di flusso che crei la tabella pitagorica dei numeri sino a 10.

### *Esercizio 5*

Realizzare un diagramma di flusso che, date le coordinate degli estremi di due rettangoli, trovi se si sovrappongono.

### *Esercizio 6*

Realizzare un diagramma di flusso che, data una parola trovi la sua speculare.

### *Esercizio 7*

Realizzare un diagramma di flusso che, dato un numero intero, trovi se è primo.



### *Esercizio 8*

Realizzare un diagramma di flusso che, dato un numero intero, trovi tutti i suoi divisori.

### *Esercizio 9*

Realizzare un diagramma di flusso che, date le coordinate degli estremi di due segmenti, trovi se sono tra loro paralleli

### *Esercizio 10*

Realizzare un diagramma di flusso che emuli il gioco del tris (conosciuto anche come Tic Tac Toc).



## INDICE

- 3.0 PSEUDO-CODICE E PSEUDO-LINGUAGGIO
- 3.1 CONCETTO DI TIPO E DI DICHIARAZIONE DI DATO
- 3.2 DISTINZIONE TRA TIPI DI DATI PRIMITIVI E STRUTTURATI
- 3.3 GLI OPERATORI
- 3.5 TIPI DI DATI PRIMITIVI PREDEFINITI
  - 3.4.1 TIPO INTEGER O INT
  - 3.4.2 TIPO REAL O FLOAT
  - 3.4.3 TIPO BOOLEAN O BOOL
- 3.5 STRUTTURE DI DATI
  - 3.5.1 L' ARRAY
  - 3.5.2 IL RECORD
  - 3.5.3 L'INSIEME
- 3.6 RAPPRESENTAZIONE DELLE STRUTTURE DI DATI DI TIPO: ARRAY, RECORD ED INSIEME
  - 3.6.1 RAPPRESENTAZIONE DI UN ARRAY
  - 3.6.2 RAPPRESENTAZIONE DEL RECORD
  - 3.6.3 RAPPRESENTAZIONE DELL'INSIEME
- 3.7 INTRODUZIONE ALLE STRUTTURE DI DATI DINAMICHE
  - 3.7.1 FILE SEQUENZIALI
  - 3.7.2 PUNTATORI
  - 3.7.3 LISTE O CODE
  - 3.7.4 ALBERI BINARI
  - 3.7.5 ALBERI ORDINATI
  - 3.7.6 GRAFI
- 3.8 ESERCIZI



### 3.0 PSEUDO-CODICE E PSEUDO-LINGUAGGIO

Nel corso della spiegazione sui i vari tipi di dati non faremo riferimento ad alcun specifico linguaggio strutturato esistente, al fine di non perdere di vista l'universalità di questi concetti basilari di programmazione strutturata.

Useremo dei simbolismi che verranno spiegati al momento della loro introduzione.

Esempi (il loro significato sarà spiegato più avanti):

$x : T$

$x = y$

$x := y$

$x[i]$

$x.s$

...

...

...



### 3.1 CONCETTO DI TIPO E DI DICHIARAZIONE DI DATO

Come abbiamo detto nel capitolo precedente: *le informazioni che vengono rese disponibili all'elaboratore consistono in un insieme di "dati", relativi al mondo reale, dai quali si debbano derivare i risultati desiderati.*

Si pone adesso in evidenza il problema di definire "cosa" e "come" sono i "dati".

In matematica è abitudine classificare le variabili rispetto ad alcune loro caratteristiche importanti. Si fanno chiare distinzioni tra variabili reali, complesse e logiche, oppure tra variabili che rappresentano valori individuali, o insiemi di valori, oppure tra frazioni, tra funzioni, ecc.

Questa stessa nozione di classificazione è egualmente importante, se non di più, nell'elaborazione dei "dati".

Nel corso delle spiegazioni ci atterremo al principio che: ***tutte le costanti, le variabili, le espressioni e le funzioni appartengono ad un certo "tipo di dato".***

Tale "**tipo**" caratterizza l'insieme dei valori al quale appartiene una costante, o di una variabile o di un'espressione o i valori che una funzione può produrre.

All'interno di un programma la regola, quasi ovunque accettata, stabilisce che il tipo associato sia reso esplicito con una "**dichiarazione**" della costante, della variabile o della funzione. La "dichiarazione" deve precedere testualmente l'applicazione della costante, variabile o funzione.

Questa regola appare particolarmente sensata se si considera che un compilatore deve fare una scelta di rappresentazione dell'oggetto, nella memoria dell'elaboratore. Evidentemente, la quantità di memoria allocata ad una variabile deve essere scelta rispettando la dimensione dell'intervallo di valori che essa potrà assumere. Quando tale informazione è nota al compilatore, la così detta "allocazione dinamica" può essere evitata (su questo concetto si ritornerà più avanti).





Le caratteristiche primarie del concetto di “**tipo di dato**” sono:

- un tipo di dato determina l'insieme dei valori al quale appartiene una costante, o i valori di una variabile o di un'espressione che può assumere, o che una funzione o un operatore possono produrre;
- il tipo del valore rappresentato da una costante, da una variabile o da un'espressione può derivare dalla sua forma, o dalla sua dichiarazione, senza necessità di eseguire il processo di computazione;
- ogni operatore ed ogni funzione accettano argomenti di un tipo fissato, e producano risultati di un tipo fissato; se un operatore ammette argomenti di tipi diversi (per es: l'operatore somma “+” viene utilizzato per sommare sia numeri interi, sia numeri reali), allora il tipo del risultato può essere determinato mediante regole specifiche del linguaggio adottato.

In funzione di queste caratteristiche un compilatore può usare tali informazioni di “tipo di dati” per controllare la compatibilità, e la legalità dei vari costrutti.

Per esempio, l'assegnamento di un valore booleano (logico) ad una variabile aritmetica (reale), può essere rilevato senza eseguire il programma.

Questo tipo di ridondanza nel testo del programma è estremamente utile per agevolare lo sviluppo degli algoritmi, e deve essere considerato uno dei vantaggi principali dei buoni linguaggi ad alto livello, rispetto al codice macchina, o al codice assembler senza tipi <sup>7</sup>. Evidentemente, alla fine i dati verranno rappresentati sotto forma di cifre binarie, indipendentemente dal fatto che il programma sia stato inizialmente concepito in un linguaggio ad alto livello dotato di “tipi di dati”, oppure in codice assembler senza “tipi di dato”.

Dal punto di vista dell'elaboratore, la memoria è un ammasso di bit, senza struttura apparente. Ma è proprio la struttura astratta che, da sola, permette ai programmatori umani di dare un significato all'ammasso di bit che costituiscono la memoria stessa.

---

<sup>7</sup> Vedi Appendice A: Gerarchia di astrazioni



### 3.2 DISTINZIONE TRA TIPI DI DATI PRIMITIVI E STRUTTURATI

Un linguaggio di programmazione dispone di tipi di dati predefiniti (detti anche “*primitivi*”), che nella maggior parte dei casi, servono per definire dei “nuovi tipi”.

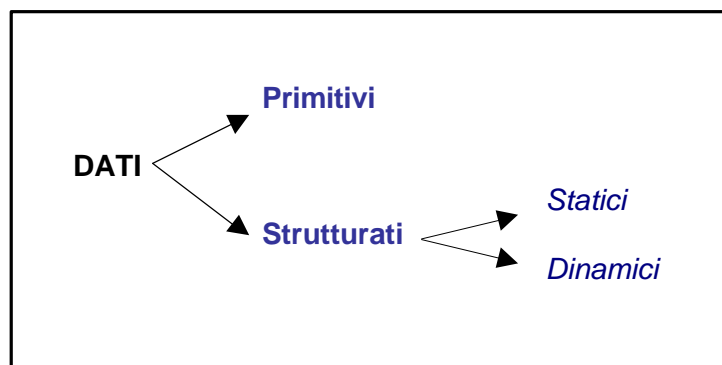
I valori, che appartengono ad un “nuovo tipo” definito in questo modo, sono di solito chiamati “*dati strutturati*” e sono agglomerati di valori componenti, appartenenti ai “*tipi costituenti*”.

Poiché i “tipi costituenti” potrebbero essere, a loro volta, strutturati ottenendo, così, intere gerarchie di tipi.

A tal punto si introduce il concetto di “**cardinalità di un dato**”.

**Definizione:** In numero dei valori distinti che appartengono ad un tipo  $T$  viene detto cardinalità di  $T$ . La si indica con la seguente annotazione  $x:T$

La cardinalità fornisce la misura della quantità di memoria necessaria per memorizzare una variabile  $x$ , di tipo  $T$ .



Classificazione dei dati



### 3.3 GLI OPERATORI

Le variabili ed i tipi di dati vengono introdotti nei programmi per poter essere usati nella computazione. A questo scopo, un insieme di “operatori” deve essere disponibile.

Come per i tipi di dati, i linguaggi di programmazione offrono un certo numero di “*operatori primitivi*” predefiniti, e dei *metodi di strutturazione*, con i quali definire operazioni composte in termini di sequenze di più operatori primitivi<sup>8</sup>.

Gli operatori primitivi (o fondamentali) più importanti sono:

- **il confronto**: per verifica di eguaglianza e di ordinamento
- **l'assegnamento**: per stabilire l'identità.

La differenza fondamentale tra questi due operatori sarà meglio definita più avanti nel testo.

Utilizzando una notazione tecnica (pseudo-linguaggio) i due operatori si indicheranno come segue:

- *confronto* (es. verifica di uguaglianza)  $x = y$
- *assegnamento* ad  $x$   $x := y$

Oltre a questi operatori esistono altri operatori fondamentali detti “*operatori di trasferimento*”. Essi associano tipi di dati ad altri tipi di dati e sono particolarmente importanti in relazione ai tipi strutturati.

A questa classe di operatori primitivi si aggiungono:

- operatori aritmetici (somma, sottrazione, moltiplicazione, divisione)
- operatori logici (AND, OR, NOT)

---

<sup>8</sup> L'attività di composizione delle operazioni viene spesso considerata il cuore della programmazione.



### 3.4 TIPI DI DATI PRIMITIVI PREDEFINITI

I tipi primitivi predefiniti sono quei tipi direttamente disponibili. Essi comprendono i numeri interi, i valori logici ed un insieme di caratteri che possono essere stampati<sup>9</sup>.

Tali tipi sono rappresentati dai seguenti identificatori:

- *integer* o *int* : numeri interi
- *real* o *float* : numeri reali
- *boolean* o *bool* : valori logici
- *char* : set di caratteri

#### 3.4.1 TIPO INTEGER O INT

E' costituito da un sottoinsieme dei numeri interi, la cui dimensione può variare da elaboratore all'altro.

Si assume che tutte le operazioni effettuate su dati di questo tipo, siano esatte e corrispondano alle usuali leggi dell'aritmetica.

La computazione verrebbe interrotta, se un risultato dovesse essere fuori dal sottoinsieme.

Gli operatori predefiniti sono le quattro operazioni fondamentali dell'aritmetica:

- somma (+)
- sottrazione (-)
- moltiplicazione (\*)
- divisione (/)<sup>10</sup>

<sup>9</sup> Vedi Appendice B: CARATTERI ASCII

<sup>10</sup> Questa operazione produce un numero intero, ignorando l'eventuale resto in modo tale che, per una coppia di numeri  $t$  ed  $s$  sia:

$$t-s < (t / s)*s \leq t$$

dove l'operatore modulo è definito in termini della divisione, mediante l'equazione:

$$(s / t)*t + (s \bmod t) = s$$

dove:  $(s / t)$  è il quoziente intero della divisione di  $t$  per  $s$   
 $(s \bmod t)$  è il resto.



### 3.4.2 TIPO REAL O FLOAT

Questo tipo individua l'insieme dei numeri reali.

L'aritmetica su valori interi considera i numeri dopo la virgola. L'accuratezza del risultato è funzione dei numeri decimali considerati; ciò implica la presenza di un errore di arrotondamento. Questa particolarità rappresenta la principale distinzione *tra tipo integer e tipo real*.

Gli operatori predefiniti sono le quattro operazioni fondamentali dell'aritmetica:

- somma (+)
- sottrazione (-)
- moltiplicazione (\*)
- divisione (/)

### 3.4.3 TIPO BOOLEAN O BOOL

I valori di tipo boolean ( o Bool) sono rappresentati dagli identificatori *true* (vero) e *false* (falso).

Gli operatori booleani servono a svolgere funzioni logiche di congiunzione (OR), di unione (AND) e di negazione (NOT).

Utilizzando un pseudo-linguaggio gli operatori logici vengono rappresentati come segue:

<b>operatore</b>		<b>simbolo</b>
congiunzione	(OR )	Ú
unione	(AND)	Ù
negazione	(NOT)	Ø



S	T	$S \cup T$	$S \cap T$	$\emptyset S$
True	True	True	True	False
True	False	True	False	False
False	True	True	False	True
false	False	False	False	True

Il risultato di un confronto può essere assegnato ad una variabile, o può essere usato come operando di un operatore logico, in un'espressione logica.

Per esempio, siano date due variabili logiche  $s$  e  $t$  e le variabili intere  $a$ ,  $b$  e  $c$  tali che:

$a = 10$        $b = 20$        $c = 30$

$s := a = b$       (<sup>11</sup>)

$t := (a < b) \wedge (b \leq c)$       (<sup>12</sup>)

$s = false$        $t = true$

<sup>11</sup> Alla variabile  $s$  viene assegnato il valore booleano nato dal confronto della relazione  $a=b$

<sup>12</sup> Alla variabile  $t$  viene assegnato il valore booleano nato dal confronto della relazione logica AND



### 3.4.4 TIPO CHAR

Il tipo predefinito *char* comprende un insieme di caratteri che possono essere stampati. Da evidenziare il fatto che non esiste un insieme di caratteri universalmente impiegato in tutti i sistemi di elaborazione. Probabilmente l'insieme di caratteri più impiegato è quello definito dall'Organizzazione Internazionale per gli Standard (ISO), ed in particolare la versione americana ASCII (American Standard Code for Information Exchange).

L'insieme ASCII consiste in 95 caratteri di stampa e 33 caratteri di controllo.

Per progettare algoritmi, che utilizzano caratteri (cioè valori di tipo *char*) indipendenti dal sistema di elaborazione utilizzato, bisogna assumere come certe le seguenti proprietà:

1. il tipo *char* contiene 26 lettere romane, le 10 cifre arabe, ed alcuni caratteri grafici, come i simboli per la punteggiatura
2. i sottoinsieme delle lettere e delle cifre sono ordinati e coerenti:

$$( 'A' \leq x ) \wedge ( x \leq 'Z' ) \equiv x$$

$$( '0' \leq x ) \wedge ( x \leq '9' ) \equiv x$$

3. il tipo *char* contiene un carattere di spazio invisibile, che può essere usato come separatore.



### 3.5 STRUTTURE DI DATI

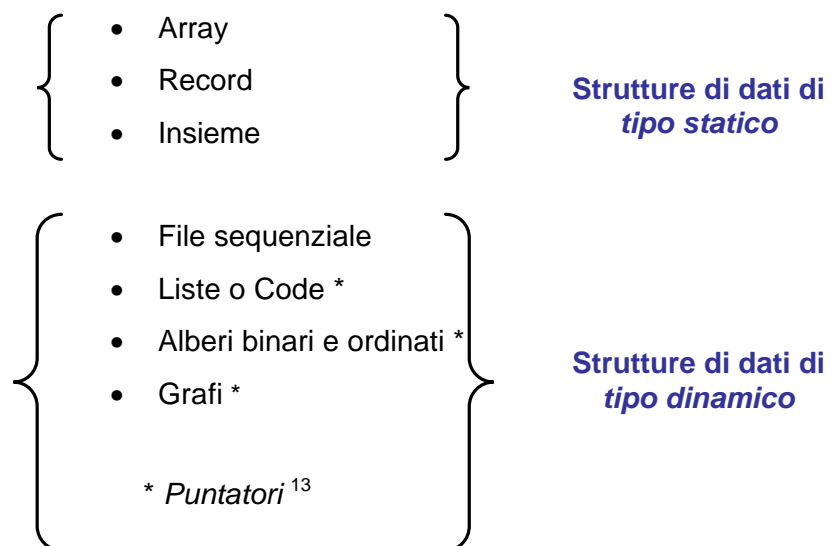
Il metodo più generale per ottenere tipi strutturati è quello di unire, in un composto, elementi di tipi arbitrari, che potrebbero essi stessi essere strutturati.

Per comprendere meglio il concetto di “tipi strutturati” potremmo prendere ad esempio i *numeri complessi* in campo matematico, essi sono composti da due numeri reali.

La stessa nozione di *coordinata cartesiana* di un punto rappresenta un tipo strutturato (si anno due numeri reali per indicare due distanze da due assi tra loro ortogonali).

Qui di seguito vengono elencate delle tipologie di “dati strutturati” composti da un insieme di “dati primitivi”, che tra loro possono essere omogenei o eterogenei.

Tra i principali dati strutturati si annotano:



<sup>13</sup> Il puntatore non è una struttura ma è uno strumento indispensabile per il controllo delle strutture di dati dinamici.





### 3.5.1 L' ARRAY

Un array è una struttura omogenea: esso consiste in componenti dello stesso “*tipo*”, detto tipo base.

L'array viene anche indicato come una struttura ad “*accesso casuale*”: tutte le sue componenti possono essere selezionate a caso, essendo egualmente accessibili.

Per poter individuare una componente individuale, il nome dell'intera struttura viene accostato da un “*indice*”, che seleziona il componente.

Per definire un array di tipo A, bisogna:

1° definire gli elementi primitivi costituenti<sup>14</sup>:

Type T	es: type real (tipi di dati di cui sarà composto l'array)
Type I	es: type integer (definizione dell'indice)

2° definizione della struttura array:

**Type A = Array [ I ] of T**    es: Array[ I ] of real

Data una variabile x, di tipo A, l'array viene rappresentato mediante il nome dell'array, con di seguito la corrispondente componente dell'indice *i*:

**x [i]<sup>15</sup>      x := A(n<sub>1</sub>, n<sub>2</sub>, n<sub>3</sub>, ..., n<sub>10</sub>)      x<sup>1</sup> Type A**

Il modo comune di operare con gli array, specialmente con quelli di grandi dimensioni, è di aggiornare in modo selettivo, singole componenti, piuttosto ricostruire valori strutturati completamente nuovi. Ciò si realizza considerando una variabile array come un vettore di variabili componenti e permettendo l'assegnamento alle componenti selezionate.

Es: x[10]                      x[2]:=0.250

<sup>14</sup> Passo implicito in quanto i tipi elementari sono definiti dall'implementatore del linguaggio o dell'elaboratore.

<sup>15</sup> Simbologia per rappresentare l'elemento i-esimo dell'array.



Sebbene l'aggiornamento selettivo faccia cambiare valore ad una sola componente, da un punto di vista concettuale si deve considerare cambiato l'intero valore strutturato.

Per quanto concerne “l'indice” di un array bisogna precisare che deve appartenere ad un “tipo scalare” (ossia numeri naturali).

Gli indici così definiti risultano calcolabili: un'espressione può sostituire una costante che rappresenta un indice. Questo aspetto non solo fornisce un potente strumento di programmazione, ma, nello stesso tempo, è causa di un tipo frequente di errori: il valore risultante potrebbe essere fuori dall'intervallo specificato come campo di variabilità degli indici dell'array<sup>16</sup>.

Essendo l'array una struttura composta emerge il problema di determinare i principi da seguire per effettuare l'ordinamento quando bisogna eseguire operazioni comparative. L'ordinamento naturale di due array viene determinato rispetto alle prime due componenti corrispondenti diverse, con minimo indice.

*Esempio:* dati due array  $t$  ed  $s$ , la relazione  $t > s$  è vera se e solo se esiste un indice  $i$  tale che:

$$t[i] > s[i] \quad \text{e} \quad t[j] = s[j] \quad \text{per} \quad j < i$$

$$(3,5,6,23,74) > (3,5,6,23,34)$$

---

<sup>16</sup> Problema risolto all'interno di alcuni linguaggi di programmazione (es: C# detto anche C sharp)



### 3.5.2 IL RECORD

Con il termine *record* si indica un composto di dati tra loro eterogenei.

Un “tipo record”  $R$  viene definito nel modo seguente:

$$\text{type } R = \text{record } \{ \begin{array}{l} e_1 : T_1 \\ e_2 : T_2 \\ e_3 : T_3 \\ \dots \\ \dots \\ e_n : T_n \end{array} \}$$

dove  $e_n$  è l'elemento costituente ennesimo di tipo  $T_n$  (detti anche “*identificatori*”).

Un valore di tipo  $R$  è costituito da un costruttore di record  $e$ , in seguito, può essere assegnato ad una variabile di tipo  $R$ :

$$x := R(x_1, x_2, \dots, x_n)$$

dove  $x_n$  sono i valori dei tipi costituenti il record ( $T_n$ ).

Gli *identificatori* possono essere richiamati autonomamente utilizzando il comando di selezione detto “*selettore di record*” o “*operatore di campo*”, rappresentato da un punto (“.”) posto a destra della variabile e seguito dal nome dell'elemento da selezionare.

L'operatore di selezione viene applicato alle variabili di tipo record.

*Esempio 1:*

data una variabile  $x : R$  tale che  $x := R(e_1, e_2, e_3)$   
la selezione individuale come segue:  $x.e_1$   $x.e_2$   $x.e_3$

*Esempio 2 :*

a) *Definizione del prototipo:*

```
type DATA = record { giorno : 1 ÷ 31 ;  
                      mese : 1 ÷ 12 ;  
                      anno : 1969 ÷ 3000 }
```

b) *Creazione della variabile di tipo DATA:*  $y := DATA$

c) *Assegnazione di valori alla variabile y:*  $y.giorno := 4 ;$   
 $y.mese := 9 ;$   
 $y.anno := 2000 ;$



### 3.5.3 L'INSIEME

Un'altra struttura fondamentale di dati è l'*insieme*.

Essa si definisce come l'insieme dei possibili valori che una variabile  $x$  di tipo  $T_0$  può assumere, dove  $T_0$  è un sottoinsieme di  $T$ .

**type**  $T_0 = \text{set of } T$                       dove  $T = (T_0, T_1, T_2, \dots, T_n)$

Esempi:

type MAIUSCOLE = set of char    ( $A \div Z$ )

type MINUSCOLE = set of char    ( $a \div z$ )

type NUMERI            = set of char    ( $0 \div 9$ )



### 3.6 RAPPRESENTAZIONE DELLE STRUTTURE DI DATI DI TIPO: ARRAY, RECORD ED INSIEME

L'essenza dell'uso delle astrazioni, nell'ambito della programmazione, è di poter concepire, studiare e verificare un programma in base alle leggi che governano le astrazioni stesse.

Avere una rappresentazione dei dati significa trovare una corrispondenza tra le strutture astratte e la memoria dell'elaboratore.

In prima approssimazione si può pensare la memoria dell'elaboratore come un array di singole celle di memoria, chiamate “parole”<sup>17</sup>. Gli indici delle parole sono chiamati “indirizzi”. La grandezza delle “parole” e degli “indirizzi” variano da un elaboratore all'altro<sup>18</sup>.

#### 3.6.1 RAPPRESENTAZIONE DI UN ARRAY

La rappresentazione di una struttura di tipo array è una corrispondenza tra l'array astratto, che ha componenti di tipo A, e una memoria, che è un array di componenti di tipo “parola”.

La corrispondenza deve essere tale che il calcolo degli indirizzi di memoria delle componenti dell'array sia:

$$i_j = i_0 + j * s$$

dove:  $i_j$  è l'indirizzo di memoria della j-esima componente

$j$  numero d'ordine della componente presa in considerazione

$s$  è il numero di “parole” occupate da una componente dell'array

Gli elementi di un array sono disposti in modo attiguo e sequenziale.

<sup>17</sup> La “parola” è, per definizione, la più piccola unità di memoria individualmente accessibile.

<sup>18</sup> Il logaritmo della cardinalità ( od ordine di grandezza) della “parola” viene chiamato “dimensione della parola”, perché rappresenta il numero di bit formanti una cella di memoria.



Poiché la “parola” è la più piccola unità di memoria accessibile è consigliabile che  $s$ , dell'array astratto, sia un numero intero.

Se  $s$  risultasse non esserlo (caso molto frequente!), allora verrebbe automaticamente arrotondato in eccesso al numero intero più prossimo  $[s]$ .

Questa operazione tecnicamente la si indica come “riempimento della memoria”

Come risultato finale si avrebbe uno spreco di memoria pari a  $[s]-s$ :

$$M_{\text{inutilizzata}} = [s] - s$$



Riempimento della “parola”. L'area puntellata rappresenta la quantità inutilizzabile.

Il *fattore reale di utilizzo* della memoria  $U$  è il rapporto tra la minima quantità necessaria a rappresentare una struttura e quella effettivamente usata:

$$U = s / [s]$$

Poiché chi programma deve mantenere l'utilizzo della memoria il più possibile vicino a  $U=1$ , e poiché accedere a parti di parole è un processo scomodo e relativamente inefficiente è necessario ricercare dei compromessi in quanto:

- 1) Il riempimento diminuisce l'utilizzo effettivo della memoria;
- 2) Omettere il riempimento può comportare un inefficiente accesso parziale alle “parole”;
- 3) L'accesso parziale alle parole può comportare l'espansione del codice (riferito al programma compilato), annullando, così, il guadagno ottenuto con l'omissione del riempimento.



I punti 2) e 3) sono gestiti automaticamente dai compilatori che utilizzano sempre automaticamente il riempimento.

Osservando il fattore di utilizzazione della memoria si possono trarre delle considerazioni:

- a)  $U > 0.5$  quando  $s > 0.5$
- b) Se  $s \leq 0.5$  il fattore di utilizzo può essere aumentato mettendo più di una componente dell'array in ogni parola. Questa tecnica è detta "*compressione*" dei dati.

$$U_{\text{compresso}} = n*s / [n*s]$$



Lunghezza della parola

*Compressione di 7 elementi all'interno di una "parola".  $n=7$*

Per accedere alla *i-esima* componente di un array compresso, è necessario calcolare l'indirizzo della parola *j*, presso la quale è localizzata la componente desiderata, e la posizione *k* della componente all'interno della parola.

Questa caratteristica è apprezzabile nei calcolatori che hanno una "parola" grande e possono accedere a porzioni di essa in modo relativamente conveniente, in quanto l'accesso avviene compiendo una sola operazione.



### 3.6.2 RAPPRESENTAZIONE DEL RECORD

I record sono memorizzati nella memoria di un elaboratore semplicemente ponendo le loro componenti in celle adiacenti.

L'indirizzo di una componente (detta *campo*)  $e_i$  relativo all'indirizzo di origine del record  $x$ , viene chiamato *offset* (distanza)  $k_i$  della componente:

$$k_i = s_1 + s_2 + s_3 + \dots + s_{i-1}$$

dove  $s_{i-1}$  è la dimensione della  $i-1$ esima componente<sup>19</sup>.

La generalità di una struttura record non permette l'introduzione di una funzione lineare per il calcolo degli offset dei suoi elementi.

Questo è il motivo per il quale le componenti di un record possono essere selezionate solo mediante identificatori fissati.

Quando diverse componenti di un record possono essere poste nella stessa "parola" di memoria, si può prendere in considerazione il problema della compressione, come descritto nel paragrafo precedente.



Rappresentazione di un record compresso all'interno di una parola.

<sup>19</sup> La dimensione è espressa in "parola".





### 3.6.3 RAPPRESENTAZIONE DELL'INSIEME

Un insieme può essere rappresentato, nella memoria di un elaboratore mediante una *funzione caratteristica*  $F_c$ .

Questa non è altro che un array di valori logici, la cui componente  $i$ -esima specifica la presenza o l'assenza del valore  $i$ -esimo nell'insieme.

La dimensione dell'array è determinata dalla cardinalità del "tipo" dell'insieme.

Esempio:

Si supponga di considerare esclusivamente i numeri da 0 a 9 che costituiscono un nostro tipo di dato T.

Un insieme del tipo T può essere il seguente:

$$s=[1,5,7,9]$$

la funzione caratteristica sarà un array di cardinalità pari a 10 e rappresentata da una sequenza di valori logici *True* e *False*:

$$F_c = (F, T, F, F, F, V, F, V, F, V)$$

Questa sequenza di valori logici saranno rappresentati all'interno della memoria del calcolatore nel modo seguente:

Bit:	0	1	0	0	0	1	0	1	0	1
Valori:	0	1	2	3	4	5	6	7	8	9

*Rappresentazione di un insieme con una stringa di bit.*

La rappresentazione degli insiemi mediante una loro funzione caratteristica ha il vantaggio che le operazioni di unione, intersezione e differenza di due insiemi possono essere implementate con operazioni logiche elementari.



### 3.7 INTRODUZIONE ALLE STRUTTURE DI DATI DINAMICHE

Una caratteristica comune alle strutture dati viste fino a qui, cioè l'array, il record e l'insieme, è che la loro *cardinalità* è *finita*. Quindi, queste strutture presentano poche difficoltà ad implementarle su qualsiasi elaboratore.

Ma la maggior parte delle strutture dei dati è caratterizzata dal fatto di avere *cardinalità infinita*. Questa differenza, rispetto alle strutture fondamentali viste, è di profonda importanza ed ha conseguenze pratiche significative.

La quantità di memoria necessaria a rappresentare un valore di un tipo di dato, non è conosciuta durante la compilazione: infatti essa può variare durante l'esecuzione del programma. Perciò si renderà necessario qualche schema di “**allocazione dinamica della memoria**”, che permetta di occuparla quando i valori si “*espandono*”, e possibilmente di recuperarla per altri usi, quando i valori si “*restringono*”.

Da ciò emerge che la rappresentazione, delle strutture di dati ad allocazione dinamica della memoria, è un problema delicato e difficile, e che la sua soluzione influenza in modo cruciale l'efficienza dei processi e l'economia della gestione della memoria.

Tra queste strutture di dati si elencano: *i file sequenziali, le liste o code, gli alberi ed i grafi.*



### 3.7.1 FILE SEQUENZIALI

Il file sequenziale, noto anche con il nome di file o di sequenza, altro non è che una serie di dati.

La notazione che viene utilizzata per definire un “tipo file” è analoga a quella vista per l'array e per l'insieme:

**type F = file of T<sub>0</sub>**

indica che ogni file di tipo *F* consiste in zero o più componenti di tipo *T<sub>0</sub>*

*Esempi:*

**type text = file of char**

**type numeri\_primi = file of int**

Questa struttura ha la caratteristica fondamentale che l'accesso ai suoi elementi può avvenire esclusivamente in modo “sequenziale”: si può accedere ad una sola specifica componente per volta. La componente viene specificata mediante una *posizione corrente* del meccanismo di accesso.

Tale posizione può essere modificata dagli *operatori per i file*<sup>20</sup>, solitamente spostandola alla componente successiva, oppure alla prima dell'intera sequenza.

Una seconda conseguenza molto importante dell'accesso sequenziale è che i processi di “scrittura” e “lettura” di una sequenza sono distinti, e non possono essere mescolati secondo un ordine arbitrario.

Un file viene costruito appendendo ripetutamente le componenti (sino alla fine di esso), e può essere ispezionato in seguito, con una lettura sequenziale. Perciò è abitudine ritenere che il file possa assumere uno stato tra due possibili:

- stato di costruzione (scrittura)
- stato di ispezione (lettura)

<sup>20</sup> Gli operatori su file vengono implementati dal programmatore stesso, anche se i compilatori moderni offrono una serie di operatori elementari su file. Ricordiamo tra questi gli operatori che consentono: la scrittura (write), la lettura (read), ecc.



Esistono alcuni supporti di memoria che permettono solo l'accesso sequenziale ad esempio i dispositivi a nastro.

Nei supporti a disco ogni pista di memorizzazione può essere trattata solo in modo sequenziale. L'accesso sequenziale è la caratteristica primaria di ogni dispositivo meccanico movibile, e anche di altri dispositivi.

Nella maggioranza delle applicazioni, i file di grosse dimensioni necessitano di qualche tipo di sottostruttura. Lo scopo di una sottostruttura è di fornire alcuni riferimenti espliciti, delle coordinate, per poter facilitare l'orientamento nella lunga sequenza di informazioni.

Anche i dispositivi di memoria esistenti forniscono mezzi per rappresentare tali punti di riferimento, ed hanno la capacità di localizzarli ad una velocità maggiore di quella che si ottiene quando viene letta l'informazione tra due punti.



### 3.7.2 PUNTATORI

La proprietà caratteristica delle strutture dinamiche, che le distingue chiaramente dalle strutture di tipo array, record ed insieme, è la loro capacità di cambiare dimensioni. Perciò è impossibile assegnare una porzione fissa di memoria ad una struttura dinamica e, come conseguenza un compilatore non può associare specifici indirizzi alle componenti di una variabile di questo tipo.

La tecnica più comunemente utilizzata per affrontare questo problema richiede *l'allocazione dinamica della memoria*: l'associazione tra specifiche locazioni di memoria e le componenti individuali della variabile.

Tale associazione avviene quando le componenti cominciano ad esistere, durante l'esecuzione del programma (run-time), piuttosto che durante la traduzione (fase della compilazione).

Quindi il compilatore alloca una porzione di memoria per mantenere *l'indirizzo* della componente allocata dinamicamente, al posto della componente stessa.

Un indirizzo di memoria può essere assegnato solo ad una speciale categoria di variabili dette **puntatori**.

Per evidenziare l'allocazione si introduce la procedura primitiva *new*. Data una variabile puntatore *p* di tipo *T*, l'istruzione:

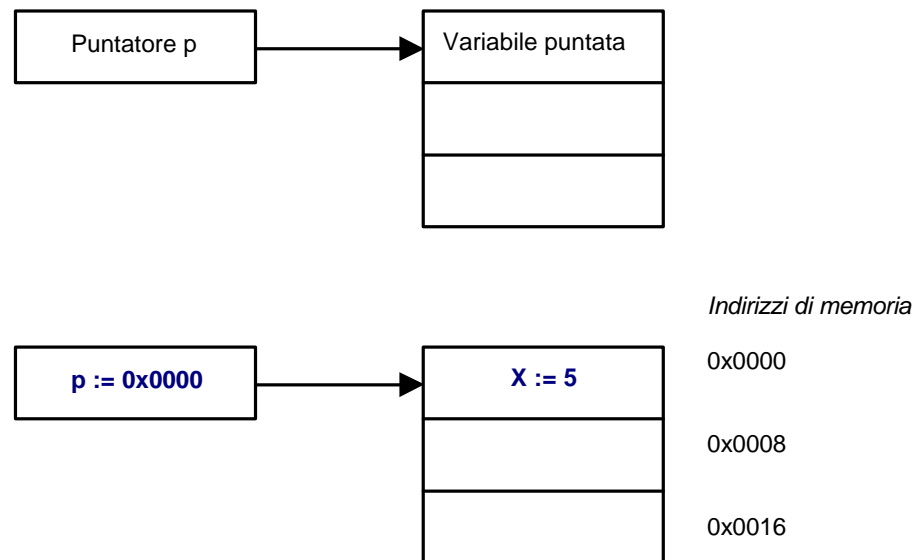
***new* (p)**

alloca effettivamente una variabile di tipo *T*, genera un puntatore di tipo *T* che fa riferimento a questa variabile ed assegna tale puntatore alla variabile *p*.

Da questo momento in poi, si potrà fare riferimento al valore del puntatore con "p"  
D'altro canto, la variabile alla quale "p" fa riferimento viene rappresentata con il simbolo "↑p". Essa è la variabile di tipo *T* allocata dinamicamente.



Graficamente questi concetti li possiamo rappresentare come illustrato nella figura sottostante:



La relazione instaurata tra puntatore e variabile puntata la si simboleggia nel modo seguente:

$$- p = x^{(21)}$$

*Il puntatore memorizza l'indirizzo di memoria in cui è memorizzato il valore assegnato alla variabile X (e non il suo valore, ossia 5).*

<sup>21</sup> La regola fondamentale dei puntatori è che devono essere dello stesso “tipo” dell’oggetto puntato.



### 3.7.3 LISTE O CODE

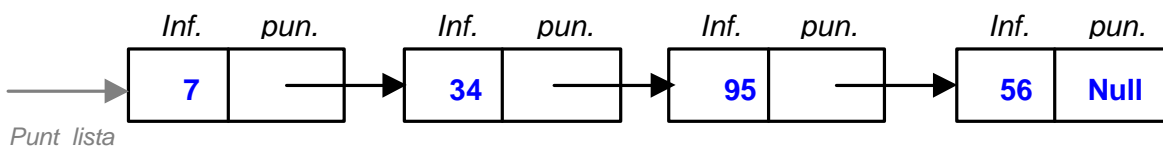
Una lista lineare è una successione di elementi omogenei che occupano in memoria una posizione qualsiasi. Ciascun elemento, appartenente alla lista, contiene un'informazione e un puntatore per mezzo del quale è legato all'elemento successivo. L'accesso alla lista avviene con il puntatore al primo elemento.

$$\text{Elemento\_lista} = \text{informazione} + \text{puntatore}$$

Il tipo lista sarà così definito:

```
type L = record { informazione :T ;  
                    puntatore   : L ;  
                }
```

Graficamente possiamo visualizzare questa struttura nel modo seguente:



Il campo puntatore dell'ultimo elemento della lista non fa riferimento a nessun altro elemento; il suo contenuto corrisponde a un segnale di fine lista.

Tale segnale dipende dal tipo di linguaggio che si utilizza nella programmazione (es: in C tale segnale è "NULL").

Una "lista vuota" non ha elementi ed è rappresentata da un *punt\_lista* che punta a Null.





E' da sottolineare che la posizione degli elementi appartenenti ad una lista **non è sequenziale**: quando si aggiunge un ulteriore elemento alla lista si deve allocare uno spazio di memoria, connetterlo all'ultimo elemento e inserirvi l'informazione relativa.

Nella struttura lineare, così definita, non esiste alcun modo di risalire da un elemento al suo antecedente.

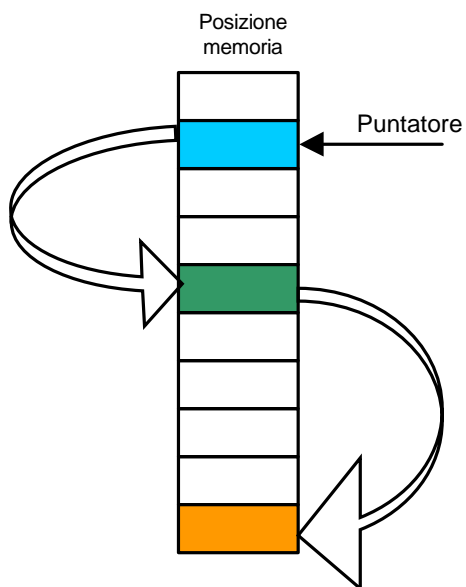
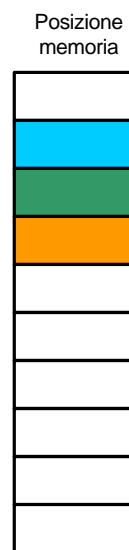
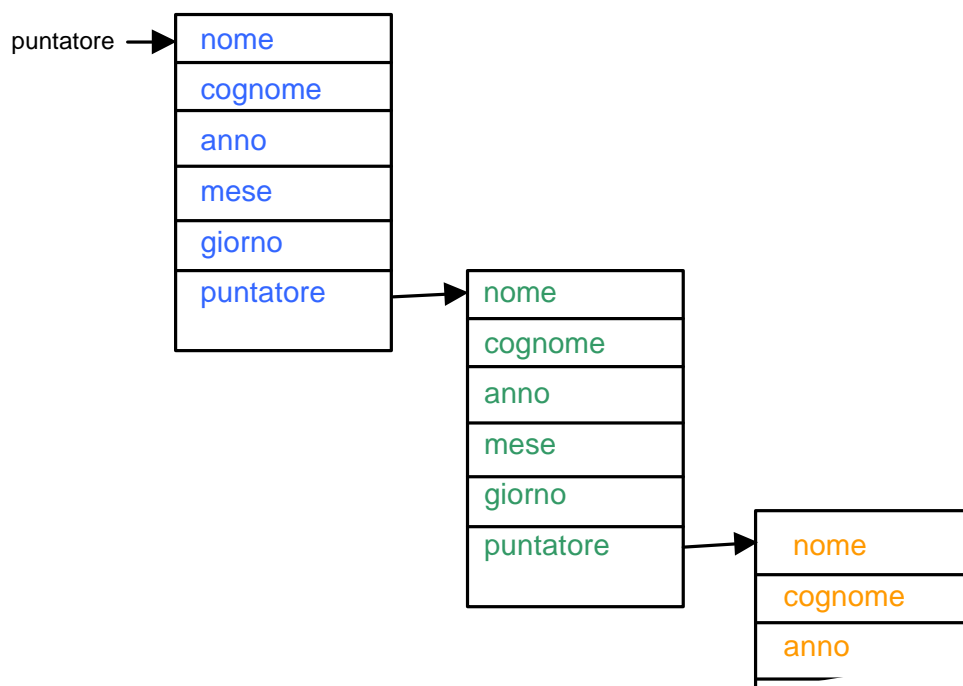
La lista si può “*scandire*” solo in ordine, da un elemento al successivo, per mezzo dei puntatori. Scandire una lista significa esaminare uno per uno i suoi elementi, dove per esaminare si può intendere: leggere, stampare, ecc.

Il segnale di “Null” è importante per verificare durante la scansione se la lista è terminata.

*Esempio:*

```
type L = record {  nome : char
                   cognome : char
                   anno : int
                   mese : int
                   giorno : int
                   puntatore : L
                   }
```



*Allocazione della memoria in una "Lista"**Allocazione della memoria in un "Array"*



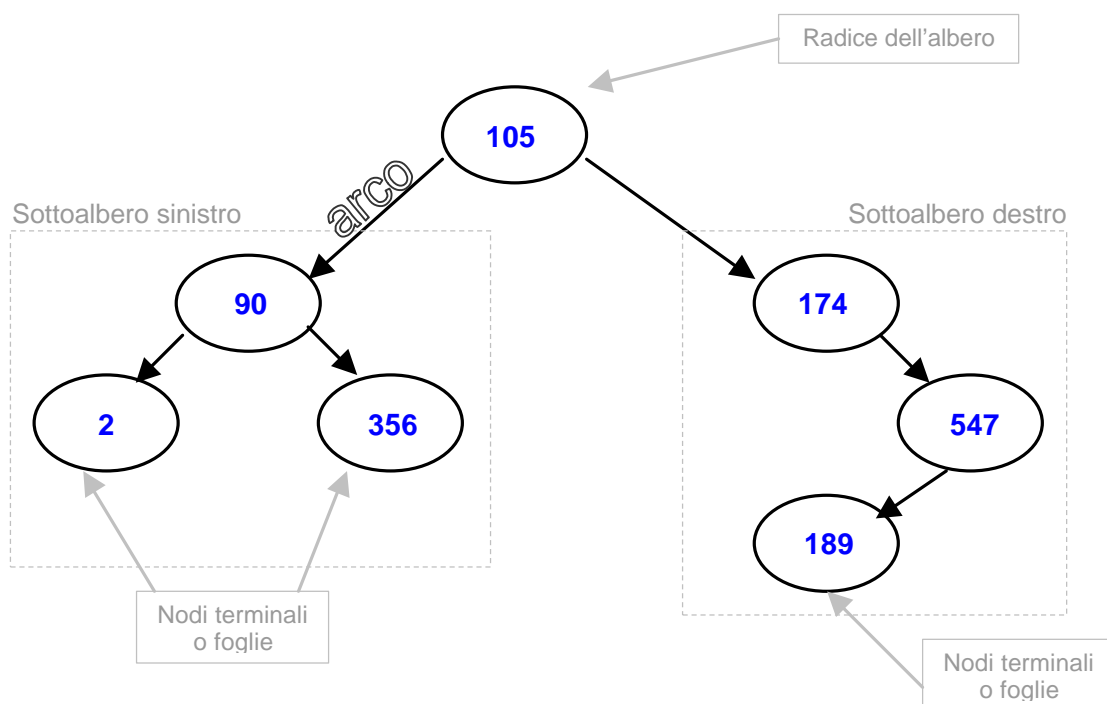
### 3.7.4 ALBERI BINARI

Le strutture dati che abbiamo fin qui esaminato sono lineari: ogni elemento della sequenza ha un successore e un predecessore, fatti salvi il primo e l'ultimo.

Esistono delle strutture in cui la relazione tra gli elementi non è lineare.

Una delle strutture dati più note è “l'**albero binario**”, definito come un insieme “ $B$ ” di *nodi*<sup>22</sup> con le seguenti proprietà:

- $B$  è vuoto oppure un nodo di  $B$  è scelto come radice
- I rimanenti nodi possono essere ripartiti in due insiemi disgiunti  $B_1$  e  $B_2$ , essi stessi definiti come alberi binari.  $B_1$  e  $B_2$  sono detti “sottoalberi” della radice<sup>23</sup>.



Esempio di albero binario

<sup>22</sup> Con il termine “nodo” si indica un *elemento* dell'albero.  
Con il termine “arco” si indica la *connessione* tra due nodi.

Con il termine di “etichetta” si fa riferimento al *valore* rappresentativo di ogni nodo  
<sup>23</sup> E' importante sottolineare che i due sottoalberi sono sempre distinti. Questa distinzione permane anche se uno dei due alberi è vuoto.

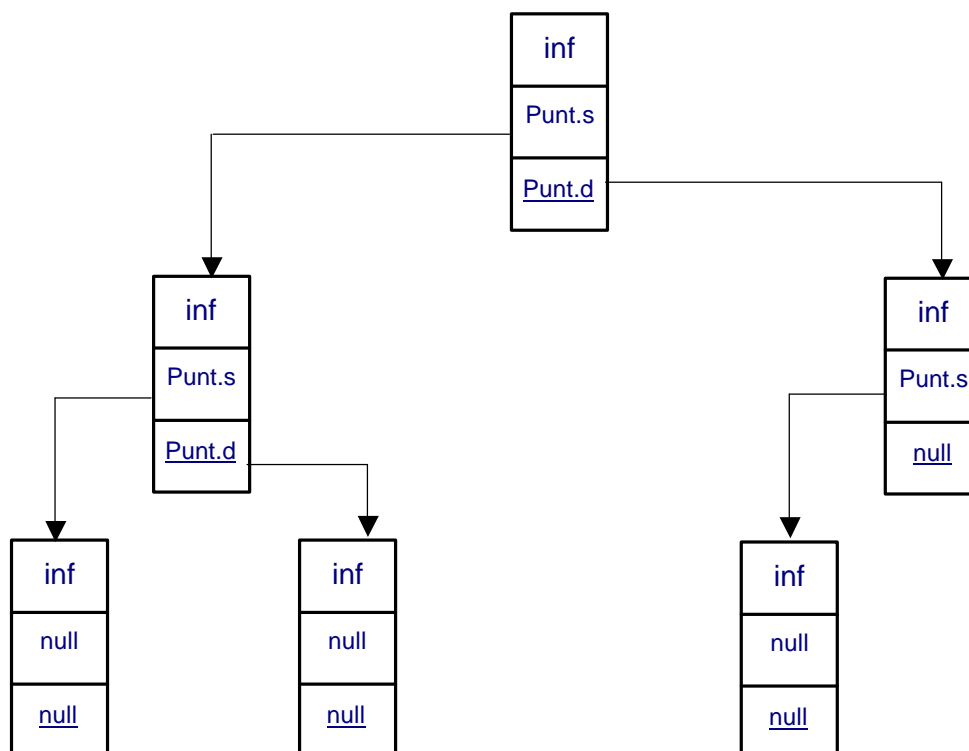


Esaminando l'immagine riportata nella pagina precedente si evince che:

- l'elemento 105 è chiamato “*radice*” dell'albero e da esso dipartono due sottoalberi;
- l'elementi 90 e 174 sono “*fratelli*” e sono rispettivamente il figlio sinistro ed il figlio destro;
- Il sottoalbero sinistro con radice 90 da due figli che non hanno alcun collegamento con altre etichette. Questi sono detti “*nodi terminali*” o *foglie*”

Il tipo albero sarà così definito come una struttura ricorsiva del tipo “nodo”:

```
type nodo = record { informazione :T ;  
                        puntatore sinistro  : nodo ;  
                        puntatore destro   : nodo ;  
                    }
```



Struttura di un albero binario



Si chiamano “**visite**” le scansioni dell'albero che portano a percorrere i vari nodi.

L'ordine in cui questo avviene distingue differenti tipi di visite:

- La visita “*in ordine anticipato*” di un albero binario viene effettuata con il seguente algoritmo:

{	<i>anticipato (radice dell'albero)</i>	}
	<i>Se l'albero non è vuoto:</i>	
	<i>Visita radice</i>	
	<i>Anticipato (radice del sottoalbero sinistro)</i>	
	<i>Anticipato (radice del sottoalbero destro)</i>	

Nel caso dell'albero in figura la visita *in ordine anticipato* dà la sequenza:

**105 , 90 , 2 , 356 , 174 , 547 , 189**

- La visita “*in ordine differito*” è descritta dal seguente algoritmo:

{	<i>differito (radice dell'albero)</i>	}
	<i>Se l'albero non è vuoto:</i>	
	<i>Differito (radice del sottoalbero sinistro)</i>	
	<i>Differito (radice del sottoalbero destro)</i>	
	<i>Visita radice</i>	

Nel caso dell'albero in figura la visita *in ordine differito* dà la sequenza:

**2 , 356 , 90 , 189 , 547 , 174 , 105**



- La visita "*in ordine simmetrico*" ha il seguente algoritmo:

{	<i>simmetrico (radice dell'albero)</i>	}
	<i>Se l'albero non è vuoto:</i>	
	<i>simmetrico (radice del sottoalbero sinistro)</i>	
	<i>simmetrico (radice del sottoalbero destro)</i>	
	<i>Visita radice</i>	

Nel caso dell'albero in figura la *visita in ordine differito* dà la sequenza:

**2 , 90 , 356, 105, 174 , 189 , 547**



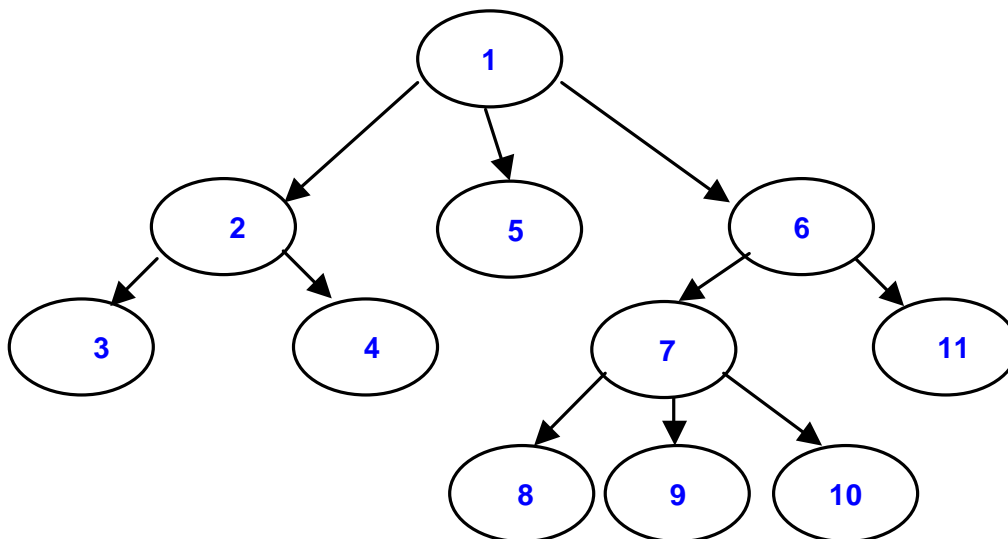
### 3.7.5 ALBERI ORDINATI

Un nodo di un albero binario può avere zero, uno o due figli. Viene spontaneo pensare a una struttura dati che superi questa restrizione: dato un insieme di uno o più nodi  $A$ , un albero è così definito:

- un nodo  $A$  è scelto come radice;
- i rimanenti nodi, se esistono, possono essere ripartiti negli insiemi disgiunti  $A_1, A_2, \dots, A_n$ , essi stessi definiti come alberi.

Si noti che la definizione data non include l'albero vuoto, come nel caso dell'albero binario; infatti si parte dall'ipotesi che "A" contenga uno o più nodi.

Di solito si è interessati ad "alberi ordinati", dove, oltre alla gerarchia insita nella struttura stessa, esiste un ordinamento tra nodi presenti a ciascun livello dell'albero.



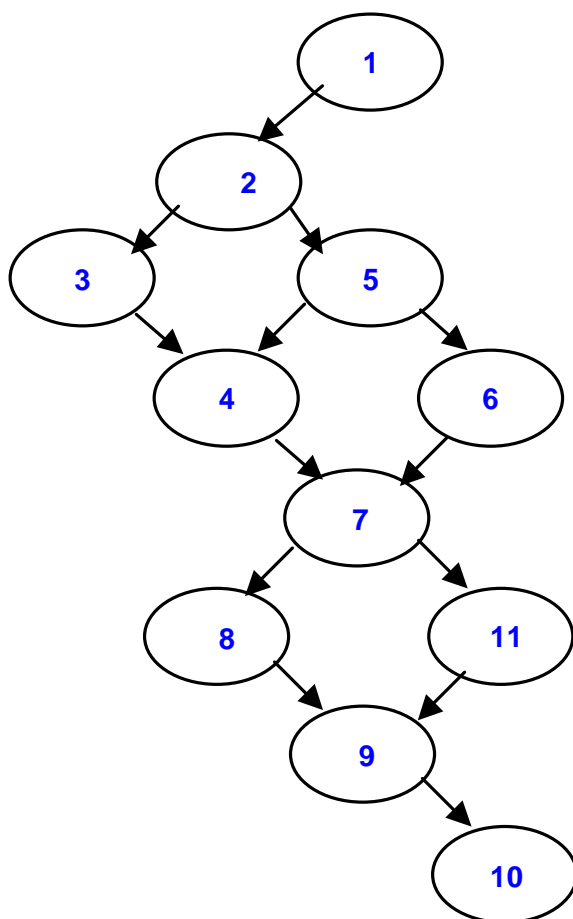
*Esempio di Albero ordinato*



Da un albero di tipo ordinato "A" di  $n$  nodi è possibile ricavare un equivalente albero binario "B" di  $n$  nodi utilizzando la regola seguente:

- la radice di A coincide con la radice di B
- ogni nodo  $b$  dell'albero B ha come radice del sottoalbero sinistro il primo figlio di  $b$  in A e come sottoalbero destro il fratello successivo di  $b$  in A

Dalla regola di trasformazione si deduce che in un albero binario, ricavato da un albero ordinato, la radice può avere soltanto il figlio sinistro.



*Albero binario ottenuto dalla trasformazione dell'albero ordinato.*

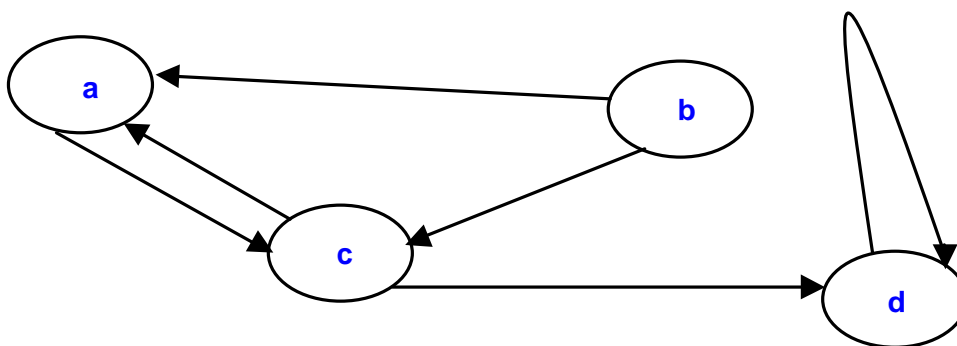


### 3.7.6 GRAFI

Un *grafo* è costituito da un insieme di nodi e un insieme di archi che uniscono coppie di nodi tali che non c'è mai più di un arco che unisce una coppia di nodi.

Il *grafo* è *orientato* quando viene attribuito un senso di percorrenza agli archi stessi: in tal caso è ammesso che tra due nodi vi siano due archi purchè *orientati* in sensi opposti; è anche possibile che un arco ricada sullo stesso nodo.

Da questa definizione è evidente come un albero sia un caso particolare di grafo, in quanto rispondono ad un insieme di regole più restrittivo.



Rappresentazione di un grafo

Un modo per memorizzare i grafi, orientati e non, è quello di utilizzare una “*matrice di adiacenza*”, in cui ogni riga e ogni colonna rappresentano un nodo.

	a	b	c	d
a	0	0	1	0
b	1	0	1	0
c	1	0	0	1
d	0	0	0	1

Nel caso di “*grafi orientati*” la **regola** è la seguente:

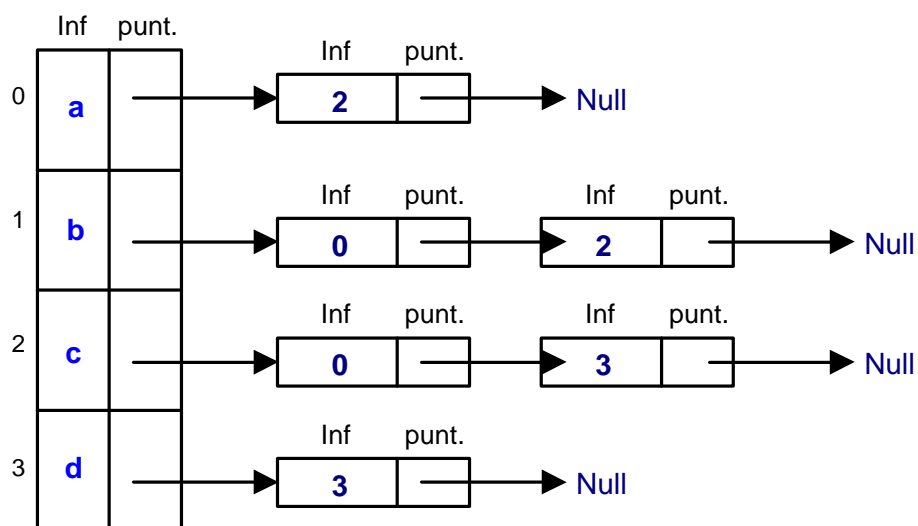
- nella matrice appare 1 se esiste un arco orientato dal nodo di riga al nodo di colonna
- nel caso di nodo chiuso su se stesso (es: nodo d) si mette sempre 1





La matrice di adiacenza provoca un notevole spreco di memoria nel caso, per esempio, che il numero “ $n$ ” di nodi sia molto elevato in confronto al numero degli archi: in ogni caso, infatti, si deve prevedere una matrice  $n \times n$ . Questa soluzione non è dunque molto flessibile.

Un'altra maniera di implementare i grafi è quella di utilizzare le liste e gli array. L'array viene utilizzato per memorizzare tutti i nodi e per ogni nodo è presente un puntatore a una lista che contiene i riferimenti ai nodi successivi.



*Grafo memorizzato mediante una lista*



### 3.8 ESERCIZI

#### *Esercizio 1*

Creare una lista che consenta la memorizzazione di una informazione concepita come segue: *nome, cognome, anni, telefono, e-mail*.

#### *Esercizio 2*

Scrivere una funzione ricorsiva per visitare in ordine differito un albero binario. Si consideri una struttura dei nodi come quella descritta nel paragrafo 3.7.4

#### *Esercizio 3*

Scrivere una funzione che dato un grafo, memorizzato in una matrice di adiacenza, e date in ingresso le etichette di due nodi, verifichi se esiste un arco che li collega.



## CAPITOLO 4

### INDICE

#### 4.1 MODULARITA' ED ASTRAZIONE

##### 4.1.1 CONCETTI DI PROGRAMMAZIONE MODULARE

##### 4.1.2 CONCETTI DI PROGRAMMAZIONE PER ASTRAZIONI

#### 4.2 COSTRUTTI LINGUISTICI

#### 4.3 DEFINIZIONI DI FUNZIONI

##### 4.3.1 METODOLOGIE DI PASSAGGIO DEGLI ARGOMENTI: PER VALORI E PER REFERENZA



## 4.1 MODULARITA' ED ASTRAZIONE

I concetti esposti sino ad ora rappresentano le basi della **programmazione strutturata**.

Questi devono essere complementati con i concetti di programmazione:

- **Modulare.** Indispensabili sia nella programmazione "*in-the-large*" (in grande), che in quella "*in-the-small*" (in piccolo). Quindi **programmare per parti**.
- **Per astrazioni.** Indispensabile nella programmazione "*in-the-small*" (in piccolo): *astrazione del dato per creare il "tipo di dato astratto" da introdurre nel programma*. Quindi **programmare per astrazioni**.

### 4.1.1 CONCETTI DI PROGRAMMAZIONE MODULARE

Nasce come metodologia di programmazione "*in the large*": un team di programmatori può suddividere il problema in sottoproblemi e, quindi, ogni programmatore si può concentrare sulla soluzione di un sottoproblema (evoluzione data dall'analisi TOP-DOWN).

*Definizione:*

La **programmazione modulare** consiste nella costruzione di programmi in *parti* dette **moduli**.

*Caratteristiche di un modulo:*

- deve essere parte **indipendente** dal resto del programma, quindi garantire una **visibilità esclusivamente locale** delle sue parti (salvo particolari eccezioni);
- deve essere **svilupabile separatamente** (compilazione e testing separate);
- avere relazioni di interazione con altre parti tramite la definizione di una **interfaccia di utilizzo**.



Queste caratteristiche conferiscono alla **programmazione modulare** le proprietà di:

- **Riusabilità;**
- **Estendibilità;**
- **Legibilità**

Un modulo può definire al suo interno diverse informazioni come: tipi, costanti, variabili, procedure<sup>24</sup> e funzioni.

La proprietà di “**località**” garantisce che tutto ciò che è definito all’interno del modulo è visibile (cioè usabile) *esclusivamente* da quel modulo, a meno che non sia specificatamente reso visibile anche all’esterno e quindi da altri moduli; quindi si può affermare che un modulo rappresenta un “**ambiente di visibilità**”.

Essendo ogni modulo definito separatamente risulta possibile eseguire la **compilazione** separatamente<sup>25</sup>.

Quindi per ottenere un buon programma modulare devono essere soddisfatte le seguenti proprietà:

- un modulo deve scambiare il **minor numero possibile di informazioni** con altri moduli, per garantire la riusabilità e l’estendibilità del codice sorgente;
- ogni modulo deve **interfacciarsi con il minor numero possibile di altri moduli**, per garantire la riusabilità e l’estendibilità;
- l’interazione tra due moduli deve essere **esplicita ed evidente**, per garantire la leggibilità.

La modularità si ottiene, generalmente, introducendo in un linguaggio di programmazione strutturata dei “costrutti” che consentono la frammentazione in moduli e la specifica della loro interazione<sup>26</sup>.

<sup>24</sup> Una procedura o funzione, in genere, stabilisce per le informazioni in esse definite, sia la visibilità che il tempo di esistenza.

Un modulo, invece, ne stabilisce solo la visibilità e delega alle procedure il compito di stabilire il tempo di esistenza.

<sup>25</sup> Le correlazioni tra moduli sono risolte staticamente durante la fase di collegamento (linking dei moduli per produrre il programma eseguibile).

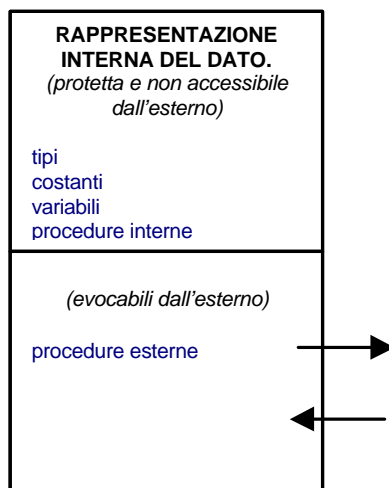


#### 4.1.2 CONCETTI DI PROGRAMMAZIONE PER ASTRAZIONI

Nasce come metodologia di programmazione *"in-the-small"*: ogni programmatore individua le astrazioni che risolvono il sottoproblema e , quindi, il team risolve il problema combinando le astrazioni sviluppate (evoluzioni dell'analisi BOTTOM-UP).

La **programmazione per astrazioni** richiede:

- la definizione di **astrazioni di dato**: realizzate attraverso un costrutto base del linguaggio usato.
- Garanzia di **protezione** e di **information hiding**<sup>27</sup>: le informazioni di una *astrazione di dato* devono essere **private** e la loro rappresentazione interna deve essere invisibile all'esterno.
- Ogni *astrazione di dato* deve definire le operazioni che sono le uniche invocabili dall'esterno e che ne rappresentano l'**interfaccia**.



Rappresentazione grafica di un astrazione di un tipo di dato

<sup>26</sup> Tra i linguaggi che utilizzano questi costrutti di modularità si elencano: ADA, Assembler 8086/88, C, C++, MESA, MODULA, MODULA2, Turbo Pascal e tutti gli altri linguaggi orientati agli oggetti.

<sup>27</sup> La proprietà di "information holding" assicura che modifiche nella rappresentazione interna del dato astratto non influenzano il resto del programma.



Quindi la **programmazione per astrazioni** consiste nella costruzione di programmi mettendo insieme delle *astrazioni di dato*.

*Definizione:*

Una **astrazione di dato** è:

- un insieme di informazioni (cioè di dati)
- unita a
- le operazioni che agiscono su di esse

Caratteristiche di una “*astrazione di dato*”:

- Protezione delle informazioni: i dati non sono accessibili dall'esterno e quindi non sono manipolabili.
- Information holding: la rappresentazione dei dati non è visibile e quindi si ha una completa indipendenza del resto del programma dalle scelte fatte.

Queste caratteristiche conferiscono alla **programmazione per astrazione di dato** le proprietà di:

- **Riusabilità;**
- **Estendibilità;**
- **Legibilità**

La programmazione per astrazione di dato si ottiene, generalmente, introducendo in un linguaggio di programmazione dei “*costrutti*” appositi<sup>28</sup>.

---

<sup>28</sup> Linguaggio con costrutti di questo tipo è il CLU.



## 4.2 COSTRUTTI LINGUISTICI

Ogni linguaggio si basa su di un “*vocabolario*” .

I suoi elementi sono solitamente chiamate “*parole*”, ma, nell’ambito della teoria dei linguaggi formali, vengono chiamati “*simboli fondamentali*”.

Una proprietà caratteristica dei linguaggi è che solo alcune alcune sequenze di parole vengono riconosciute come “*frasi*” corrette e ben formate del linguaggio, mentre altre sono considerate errate o mal formate.

Che cosa determina la correttezza di una frase?

La risposta è: la grammatica, la sintassi o struttura del linguaggio.

Infatti la “*sintassi*” è definita come l’insieme delle regole, o formule, che definiscono l’insieme delle frasi. Tale insieme di regole non solo permette di decidere se una sequenza di parole sia una frase, ma fornisce anche una struttura che è fondamentale per comprendere il significato della frase. E’ perciò chiaro che la sintassi e la semantica (cioè il significato) sono intimamente collegate.

Si consideri ad esempio la frase:

**<< Francesco studia >>**

La parola “*Francesco*” è il soggetto e “*studia*” è il predicato.

Questa frase potrebbe appartenere ad un linguaggio definito, per esempio, dalla seguente grammatica:

*<frase> ::= <soggetto> <predicato>*

dove:

*<soggetto> ::= Luca | Francesco*

*<predicato> ::= Studia | Programma*

L’idea è quindi che una frase possa essere derivata a partire dal “*simbolo iniziale*” *<frase>* mediante ripetute applicazioni di certe “*regole di sostituzione*”.





Il formalismo, o notazione, che è stato impiegato per scrivere queste regole viene chiamato: **Forma di Backus-Naur (BNF)**. Esso fu impiegato, per la prima volta, per la definizione del linguaggio ALGOL 60.

I simboli:

- *<frase>*, *<soggetto>* e *<predicato>* sono detti “*simboli non terminali*”

Le parole:

- *Luca*, *Francesco*, *studia*, *programma* sono chiamate “*simboli terminali*”

I simboli:

- *::=* e *|* sono detti “*metasimboli*” della notazione BNF

I traduttori di linguaggio hanno principalmente il compito di “riconoscere”, e non di generare, le frasi di un linguaggio.

Ciò implica che i passi di generazione, che portano ad un frase, debbano essere ricostruiti mentre la frase viene letta. Questa operazione è, in generale, molto complessa e, talvolta, persino impossibile. La sua complessità dipende fortemente dal tipo delle regole di produzione che sono state impiegate per definire il linguaggio.



### 4.3 DEFINIZIONI DI FUNZIONI

Un programma è formato da elementi connessi in modo da raggiungere un determinato scopo. Le istruzioni possono essere considerate i componenti di un programma.

Ciascuna istruzione corrisponde a un'azione elementare: ponendo le istruzioni in un determinato ordine il compilatore interpreta le frasi e consente, così, al programma di svolgere il compito cui era stato destinato dal programmatore.

Un programma può essere scomposto non oltre il limite delle singole istruzioni che lo compongono, ma è possibile aggregare gruppi di istruzioni per formare i così detti blocchi o “sottoprogrammi” o “moduli”.

I moduli si usano anche per evitare di replicare porzioni di codice sorgente: invocare un modulo significa mandare in esecuzione la porzione di codice corrispondente. Se un modulo è invocato più volte, la porzione di codice è eseguita più volte, tante quante sono le invocazioni. Il vantaggio dei moduli è appunto di consentire al programmatore di avere tante chiamate ma una sola porzione di codice.

E' possibile poi creare delle librerie, cioè delle raccolte di moduli che possono essere utilizzati senza essere a conoscenza dei dettagli implementativi<sup>29</sup>.

In C, ad esempio, i moduli o sottoprogrammi sono detti “*funzioni*”. Queste funzioni utilizzano i dati di ingresso per effettuare le elaborazioni preposte, per poi restituire esclusivamente un valore al programma chiamante.

Una funzione può essere pensata come una “*scatola nera*” che a determinati valori d'ingresso fa corrispondere un determinato valore d'uscita, senza conoscere la dinamica interna di funzionamento posta all'interno della scatola.



<sup>29</sup> E' quanto avviene, ad esempio, quando nel linguaggio C si utilizzano le funzioni come: `printf()` o `scanf()`.



Per poter utilizzare una funzione bisogna :

- dichiarare la funzione; operazione che definisce :
  - il nome della funzione stessa;
  - il tipo di dati in ingresso detti *parametri formali*;
  - il tipo di dati in uscita.
- definire il costrutto della funzione, ossia l'insieme delle istruzioni che dovranno essere eseguite quando viene invocata la funzione.

Esempio di dichiarazione:

***tipo\_ritorno Nome\_funzione (tipo par1, ..., tipo parN);***

↓                      ↓                      ↓

*double area (float , float );*

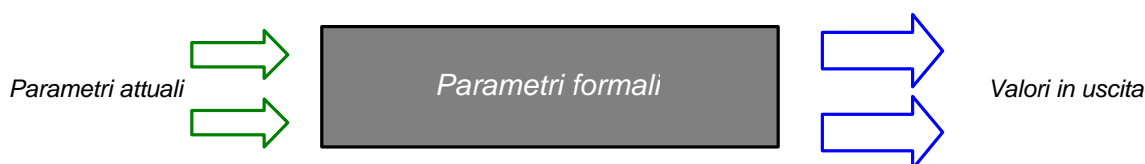
Esempio di definizione del costrutto della funzione:

```
double area (float base, float altezza)  
{  
    return base*altezza;  
}
```



#### 4.3.1 METODOLOGIE DI PASSAGGIO DEGLI ARGOMENTI: PER VALORI E PER REFERENZA

Una funzione viene invocata facendo riferimento al nome e passando ad essa una lista di parametri conformi in tipo, numero e ordine alla lista dei *parametri formali* elencata nella definizione della funzione stessa; tali parametri sono detti *parametri attuali*.



Il passaggio di tali valori può avvenire in due modi:

- **per valore;**
- **per referenza.**

**Il passaggio per valore** significa che all'atto dell'invocazione di una funzione ogni parametro formale è inizializzato con il valore del corrispondente parametro attuale. Ecco perché deve esistere una coerenza di tipo e di numero tra i parametri formali e quelli attuali.

Poiché con il passaggio dei parametri i valori dei parametri attuali sono travasati nelle locazioni di memoria corrispondenti ai parametri formali, si ha che la semantica del passaggio dei parametri è quella della inizializzazioni di variabili causando un consumo di memoria aggiuntiva sino quando la funzione non termina il proprio lavoro.

Operando in questo modo si evita di commettere operazioni sui valori attuali in quanto la funzione agisce su una loro copia (ossia i parametri formali).

**Il passaggio per referenza** invece utilizza il metodo di passare l'indirizzo di memoria in cui sono contenuti i valori attuali, consentendoci un risparmio di memoria.

Questo metodo è da utilizzarlo con molta attenzione in quanto la funzione legge i dati direttamente nel luogo in cui sono memorizzati e non su delle copie. Teoricamente la funzione potrebbe essere in grado di sovrascrivere il contenuto e quindi il programma potrebbe perdere i valori iniziali.



## INDICE

- A. GERARCHIA DI ASTRAZIONI IN UN ELABORATORE
- B. CODICE ASCII
- C. PROTOTIPO O TIPO DEL VERO PROGRAMMATORE



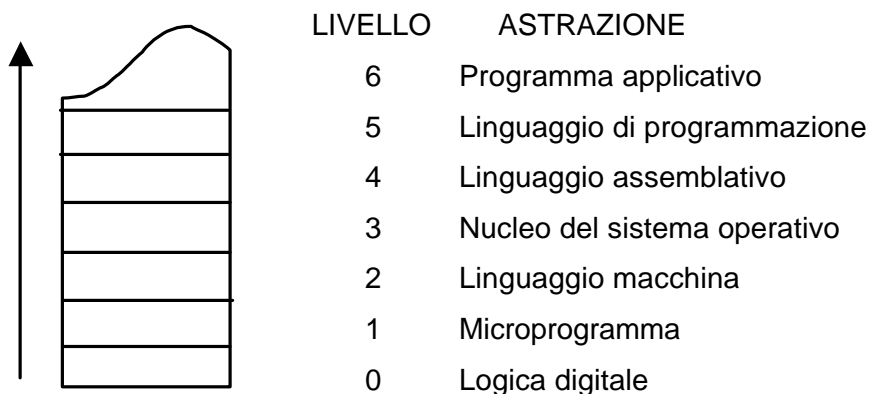
## &lt; APPENDICE A &gt;

## GERARCHIA DI ASTRAZIONE IN UN ELABORATORE

Un elaboratore, con i programmi ad esso associati, presenta una gerarchia di livelli di astrazione, chiamate **macchine virtuali**.

Ogni livello, ad eccezione del più basso, è realizzato **traducendo** o **interpretando** le sue istruzioni mediante le **primitive** fornite dai livelli inferiori.

Qui di seguito sono riportati questi livelli di astrazione:

**Livello 0: Logica digitale**

E' un'astrazione della circuiteria elettronica, realizzata mediante circuiti detti "*porte*", con uno o più "*ingressi digitali*" (rappresentati da 0 e 1) e che producono in uscita una "*funzione logica*" dei propri dati di ingresso (es: AND o OR).

**Livello 1: Microprogramma**

E' il primo livello costituito in modo prevalente da "*linguaggio*", non è posseduto da tutti i computer, si tratta di un "*insieme di passi*" usati per realizzare le istruzioni del linguaggio macchina.

**Livello 2: Linguaggio macchina**

E' costituito da istruzioni primitive sufficienti a eseguire qualsiasi programma o applicazione dei livelli più alti.

**Livello 3: Nucleo del sistema operativo**

Ha le “*primitive*” per *ordinare* e *allocare* le *risorse* del calcolatore per i diversi programmi che vengono eseguiti.

**Livello 4: Linguaggio assembler**

E' una “*rappresentazione simbolica*” delle istruzioni dei livelli inferiori. Un programma in linguaggio assembler viene tradotto in istruzioni di livello inferiore mediante un traduttore, detto “*assemblatore*”.

**Livello 5: Linguaggio di programmazione**

Consente una più rapida risoluzione di problemi di quanto consenta il linguaggio assembler in quanto dotato di istruzioni più intuitive per il programmatore. La traduzione viene effettuata dal “*compilatore*”.

Esempi di linguaggi di programmazione sono: C, C++, Pascal, Fortran, Basic, ecc.

**Livello 6: Programma applicativo**

E' il software che l'utente utilizza per interagire con l'elaboratore senza preoccuparsi della logica interna dell'elaboratore.



## &lt; APPENDICE B &gt;

## CARATTERI ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>MUL</b> (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	\$	68	44	104	&#68;	D
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	}
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	~
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_





## ASCII ESTESO

128	Ç	144	É	160	á	176	░	193	⌞	209	ƒ	225	ß	241	±
129	ù	145	æ	161	í	177	▒	194	⌟	210	π	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	⌠	211	ℓ	227	π	243	≤
131	â	147	ô	163	ú	179		196	—	212	ℓ	228	Σ	244	∫
132	ä	148	ö	164	ñ	180	†	197	‡	213	ƒ	229	σ	245	∫
133	à	149	ò	165	Ñ	181	‡	198	‡	214	ƒ	230	μ	246	÷
134	å	150	û	166	ª	182	‡	199	‡	215	‡	231	τ	247	≈
135	ç	151	ù	167	º	183	π	200	ℓ	216	‡	232	Φ	248	°
136	ê	152	—	168	¿	184	¶	201	ƒ	217	∫	233	Θ	249	·
137	ë	153	Ö	169	—	185	¶	202	‡	218	∫	234	Ω	250	·
138	è	154	Û	170	¬	186		203	ƒ	219	■	235	δ	251	√
139	ï	156	£	171	½	187	¶	204	‡	220	■	236	∞	252	—
140	î	157	¥	172	¼	188	¶	205	=	221	■	237	φ	253	²
141	ì	158	—	173	¡	189	¶	206	‡	222	■	238	ε	254	■
142	Ä	159	ƒ	174	«	190	¶	207	⌞	223	■	239	∩	255	
143	Å	192	Ł	175	»	191	¶	208	⌞	224	α	240	≡		



## &lt; APPENDICE C &gt;

## PROTOTIPO O “TIPO” DEL VERO PROGRAMMATORE

Questa appendice vuole rappresentare in tono sarcastico la figura di un prototipo o “*tipo di programmatore*”.

Prendete questa lettura in tono molto scherzoso.

Mi vorrei personalmente complimentare con l'autore di questo testo (dal sito dal quale è stato prelevato non risultava alcun nominativo)

## 1) DEFINIZIONE DEL VERO PROGRAMMATORE

Il sistema piu' rapido e sicuro per distinguere un Vero programmatore dal resto del mondo e' considerare il linguaggio che usa: il Vero Programmatore programmava in FORTRAN, mentre ora programma in C.

I mangiatori di Quiche programmano in Pascal. Da questo si deduce che sicuramente Niklaus Wirth era un mangiatore di Quiche e NON un Vero Programmatore.

Ad un Vero Programmatore non servono tutte le strutture ed i meccanismi del pascal, un Vero Programmatore puo' essere felice con un perforatore di schede o un terminale a 1200 baud, un C a standard K&R (ANSI... a che serve, il K&R e' fin troppo chiaro), ed una birra.

A proposito, Kernigan e Ritchie sicuramente erano dei Veri Uomini. probabilmente anche dei veri programmatori.

- Il Vero Programmatore processa liste in C
- Il Vero Programmatore processa numeri in C
- Il Vero Programmatore manipola stringhe in C
- Il Vero Programmatore elabora programmi di I.A. in C
- Il Vero Programmatore fa contabilita' in C
- Il Vero Programmatore crea simulatori di reti neurali in C
- Il Vero Programmatore starnutisce in C
- Il Vero Programmatore fa TUTTO in C

Se per caso il C non fosse sufficiente il Vero Programmatore lavorera' in assembler, se neppure questo fosse sufficiente allora il lavoro non e' fattibile, ma la cosa e' impossibile, un Vero Programmatore in C ed assembler puo' fare TUTTO, per definizione.



## 2) PROGRAMMAZIONE STRUTTURATA

Gli accademici negli ultimi anni hanno stabilito, dall'alto delle loro cattedre, che un programma è più facilmente leggibile se il programmatore utilizza particolari tecniche, strutture e costrutti.

Ovviamente essi non sono d'accordo su quali questi costrutti e queste tecniche precisamente siano, e perciò le loro teorie sono discordanti ed erratiche. In questo modo solo alcuni mangia-Quiche si lasciano convincere dai loro assiomi.

Un tipico lavoro del mondo reale (e non un lavoro teorico da università) è di prendere un sorgente di 100.000 o 200.000 linee e farlo andare il doppio più veloce. In questo caso qualunque Vero Programmatore vi potrà dire che la programmazione strutturata non serve a nulla, quello che in realtà serve è del talento.

Alcune rapide considerazioni del Vero Programmatore sulla programmazione strutturata:

- Il Vero Programmatore non ha paura di usare GOTO
- Il Vero Programmatore può scrivere un ciclo DO lungo 5 pagine senza fare confusione.
- Il Vero Programmatore usa i costrutti CASE basati su calcoli aritmetici, essi rendono un programma piu' divertente.
- Il Vero Programmatore scrive del codice automodificante, soprattutto se questo può salvare 20 nanosecondi all'interno di un ciclo.
- Il Vero Programmatore utilizza l'area di memoria di un codice già eseguito e che non servirà più come area di memoria per i dati, ottimizzando in questo modo lo spazio a disposizione.
- Il Vero Programmatore non ha bisogno di commenti, il codice è già autoesplicante a sufficienza.

Dopo aver parlato di programmazione strutturata si è anche parlato molto di strutture di dati. Tipi di dati astratti, stringhe, liste e chi più ne ha più ne metta. Wirth (il mangiatore di Quiche menzionato poco sopra) ha scritto un intero libro tentando di dimostrare che si può scrivere un intero programma basandosi solo sulle strutture di dati.



Come ogni Vero Programmatore sa invece l'unica struttura che serve **VERAMENTE** è l'array, dato che tutti gli altri tipi di dato altro non sono che sottoinsieme limitati di questi. e dato che sono limitati egli usa solo puntatori, soprattutto se questi rendono possibile bombare irrimediabilmente il computer, altrimenti dove starebbe il divertimento?

### 3) SISTEMI OPERATIVI

Che S.O. usa un Vero Programmatore?

UNIX? NO!, Unix e' qualcosa di simile a quello che si aspetta un vero Hacker, dato che qualunque Vero Programmatore non trova alcun divertimento nel tentare di indovinare come cavolo il comando PRINT viene chiamato questa settimana. La gente non fa lavori seri su Unix, lo usano soprattutto per fare adventure, modificare Rogue e mandarsi il tutto via UUCP.

MS-DOS?

Gia' meglio, critico quel tanto che basta, facile da modificare, se ce ne fosse necessita', facile da bombare, con tante cose sconosciute e strane. Una cosa sicuramente possiamo dire:

- Il Vero Programmatore non usa il mouse e le icone, infatti il Vero Programmatore non capisce perche' mai per compilare un programma uno debba staccare le mani dalla tastiera e cliccare su un menu quando e' tanto semplice battere:  
CL pippo.c -k -iC:\gnu\c\all -q -w -e -r +t -y +cvb +f -g +g +p =l /f /a /s

Comunque il Vero Programmatore ha una sola nostalgia: il sistema IBM OS/370. Questo era infatti il SO che qualunque Vero Programmatore vorrebbe vedere implementato su TUTTI i computer del mondo.

Un Vero Programmatore sa che se vede comparire l'errore IJK3051 basta andare a vedere nel manuale del JCL per capire cosa e' successo.

Un Grande Programmatore poi sapra' i codici a memoria, mentre un Grandissimo Programmatore potra' trovare l'errore osservando 6 mega di dump senza neppure usare un calcolatore esadecimale...

L'OS/370 e' VERAMENTE un SO potente, infatti e' possibile distruggere giorni e giorni di lavoro con la semplice pressione di un tasto. Questo incoraggia l'attenzione sul lavoro e forma una mentalita' che servira' in futuro, quando per distruggere giorni di lavoro saranno sufficienti tre tasti



#### 4) TOOL DI PROGRAMMAZIONE

Quali tool di programmazione necessita realmente un Vero Programmatore? In effetti, come detto prima sono sufficienti un terminale a 1200 baud o un lettore di schede perforate, ma anche una semplice tastiera esadecimale sarebbe gia' piu' che sufficiente.

Ma purtroppo adesso i computer non hanno piu' tastiere esadecimali, come pure non hanno piu' quei magnifici pannelli frontali pieni di lucine e tastini che facevano tanto futuro.

I primi veri programmatori sapevano a memoria l'intero settore di boot dell'hard disk, e lo potevano riscrivere a memoria ogniqualevolta che il loro programma lo rovinava. La leggenda narra che Seymore Cray (creatore del Cray I) scrisse il SO del primo CDC7600 usando il pannello frontale del computer la prima volta che questo venne acceso. Senza bisogno di dirlo Seymore era un Vero Programmatore. Uno dei migliori Veri Programmatori che abbia mai conosciuto e' un sistemista della Texas Instrument. Una volta rispose alla telefonata di un cliente a cui si era bombato il sistema durante il salvataggio del lavoro. Il Vero Programmatore rimise a posto tutto facendo scrivere le istruzioni per terminare il lavoro di I/O sul pannello frontale (allora c'erano ancora), riscrivendo i dati rovinati in esadecimale e facendosi dire i risultati per telefono. La morale della storia e' che se un tastierino ed una stampante possono far comodo un Vero Programmatore puo' arrangiarsi anche con solo un telefono. Un altro tool fondamentale e' un buon text editor. Molti dicono che il migliore sia quello della Xerox di Palo Alto, ma, come gia' detto, il Vero Programmatore non parla al suo computer attraverso un mouse.

Altri preferiscono EMACS o VI, ma in effetti il concetto di WYSIWYG (quello che vedi e' quello che ottieni) si applica ai computer malissimo, cosi' come si applica alle donne. Quello che un vero programmatore vuole e' in effetti qualcosa di piu' complesso, che implementi la filosofia del "You asked for it, you got it !!!" (YAFIYGI, avrai solo quello che chiedi).

Insomma, l'editor perfetto e' il TECO.

Alcuni hanno osservato che una linea di comandi per TECO assomiglia molto di piu' al rumore sulle linee telefoniche che ad una linea di comandi, ed in effetti uno dei giochi piu' divertenti da fare e' quello di scrivere il proprio nome sulla linea di comando e vedere cosa succede. Inoltre ogni piccolo errore avra' come risultato quello di distruggere il vostro programma, o, peggio, di introdurre subdoli errori che saranno in seguito difficilmente rintracciabili.

Per questa ragione un Vero Programmatore e' molto riluttante a editare un programma funzionante per dargli gli ultimi ritocchi. E sempre per questa ragione un Vero Programmatore trova piu' semplice fare le modifiche finali utilizzando un programma come lo Zap.

Alcuni Veri Programmatori utilizzano lo Zap stesso come editor, altri scrivono il programma direttamente in codice eseguibile, ma e' forse esagerato.



Procedendo su questa linea il risultato e' che tra il codice sorgente e quello che in effetti c'e' scritto su disco c'e' una discrepanza sempre maggiore, con il risultato che il lavoro e' sempre piu' sicuro, perche' solo un Vero Programmatore potra' lavorarci sopra in modo proficuo, nessun mangiatore di Quiche potra' fare manutenzione, minimizzando cosi' i rischi di malfunzionamenti ulteriori del programma.

Questa e' SICUREZZA.

Altri tool importanti sono le documentazioni su cui il vero programmatore basa gran parte del suo lavoro:

- Il Vero Programmatore non legge mai i manuali introduttivi, bastano ed avanzano i Reference Manual.
- Il Vero Programmatore ha imparato il C sul K&R, qualunque altro testo e' inutile e deviante.
- Il Vero Programmatore se possibile legge i manuali in lingua originale, anche se questo a volte pone dei problemi di reperibilita'.
- Il Vero Programmatore non colleziona libri di raccolte di algoritmi. Questo perche' e' piu' lento cercare l'algoritmo in 3000 pagine di manuale che scriverlo di getto.
- Il Vero Programmatore non ha bisogno di manuali sull'assembler, sono sufficienti i data sheet dei microprocessori.
- Il Vero Programmatore non scrive MAI i manuali dei programmi che fa, non ne ha il tempo materiale.

Il Vero Programmatore generalmente ha da qualche parte la documentazione completa del SO su cui lavora, pubblicata dalla casa che ha fatto il SO, ma sa che SICURAMENTE nelle 3500 pagine che in media compongono la documentazione non trovera' quello che cerca.

Se nelle vicinanze del terminale sono presenti piu' di 5 manuali ci sono delle forti probabilita' che NON sia un Vero Programmatore.

Alcuni Tool NON usati da un Vero Programmatore:

- Preprocessori di linguaggio.
- Traduttori di linguaggio.
- Full Screen Debugger a livello sorgente. Il Vero Programmatore e' in grado di capire quello che dice il Debug.
- Compilatori ottimizzanti. L'ottimizzazione del programma scritto dal Vero Programmatore e' gia' il massimo, e percio' altre modifiche non farebbero altro che peggiorare la situazione.



## 5) IL LAVORO DEL VERO PROGRAMMATORE

In generale il Vero Programmatore non fa lavori semplici come gestione di indirizzari o programmi gestionali, ecco alcuni dei lavori piu' adatti ai veri programmatori:

- Il Vero Programmatore scrive programmi per la simulazione di una guerra termonucleare per l'esercito.
- Il Vero Programmatore lavora per lo spionaggio, per decrittare le trasmissioni in cifra del nemico.
- E' in gran parte dovuto al lavoro dei Veri Programmatori se gli americani sono arrivati sulla Luna.
- Il Vero Programmatore programma i sistemi guida di satelliti e missili.
- In ogni caso il Vero Programmatore lavora su progetti molto importanti o molto ben pagati.
- Il Vero Programmatore non lavora quindi **mai per Assicurazioni, Istituti Bancari, Anagrafi e aziende simili**, se lo dovesse fare a causa di motivi contingenti, ritiene di essere indispensabile, ed elabora elenchi di bollette, polizze assicurative o estratti conto, comportandosi come se dovesse far partire la missione Apollo.

## 6) IL GIOCO

Il generale il Vero Programmatore gioca nello stesso modo in cui lavora: con i computer.

In generale lo stesso lavoro e' un gioco, ed alla fine del mese il Vero Programmatore e' sempre abbastanza stupito di ricevere un compenso per quello che, a tutti gli effetti, e' per lui un divertimento, anche se non lo dira' mai a voce alta (guai a chi di voi lo dirà al mio capo!).

Occasionalmente il Vero Programmatore uscirà dall'ufficio per prendere una boccata d'aria e farsi una birra, ecco alcuni sistemi per riconoscere un Vero Programmatore fuori dal suo posto di lavoro:

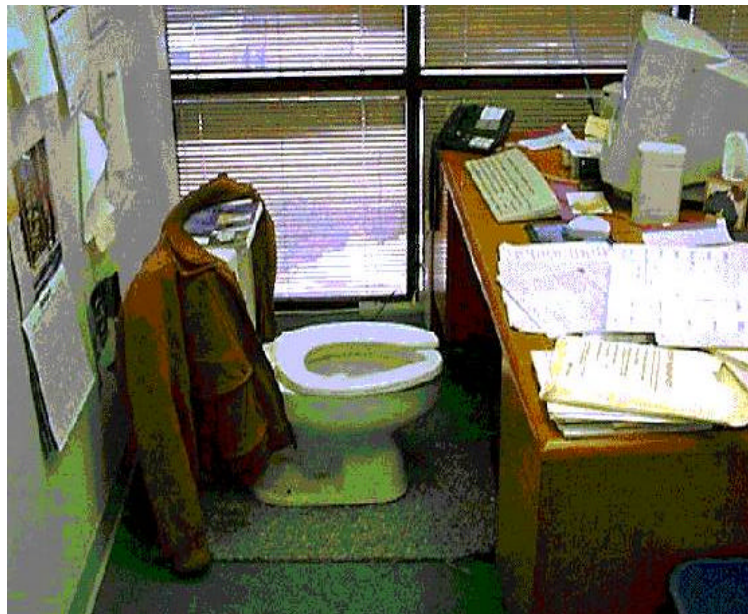
- Ad un party i Veri Programmatori sono quelli che stanno in angolo parlando di Sistemi Operativi, mentre di fianco a loro passano ragazze che si fermano, ascoltano per alcuni secondi e poi, dato che non capiscono una parola, se ne vanno.





- A volte un vero Programmatore incontra una Vera Programmatrice. Vi risparmio per decenza il racconto di come si svolgono i fatti.
- Ad una partita di football il Vero Programmatore e' quello che controlla gli schemi delle squadre basandosi su quelli disegnati dal suo programma su di un foglio 11x14.
- Sulla spiaggia il Vero Programmatore e' quello che disegna flow chart sulla sabbia.
- Durante un black out un Vero Programmatore generalmente sviene in quanto vengono a mancare i vitali afflussi di energia che gli permettono di vivere

## 7) L'HABITAT DEL VERO PROGRAMMATORE



Dal momento che un vero programmatore e', per l'azienda che lo usa, generalmente molto costoso, vediamo come fare per farlo rendere al meglio sul posto di lavoro. Il Vero Programmatore vive davanti ad uno o piu' monitor, attorno, sopra, dietro e sotto questi terminali si trovano generalmente le seguenti cose:

- I listati di TUTTI i programmi a cui il Vero Programmatore ha mai lavorato, accatastati, in ordine piu' o meno cronologico, su ogni superficie piatta disponibile intorno.





- Sei o piu' tazze di caffe', quasi sempre fredde, ed alcune con alcuni mozziconi di sigaretta galleggianti.

- Attaccato al muro c'e' un ritratto di Spock con in mano l'enterprise stampato con una vecchia stampante a margherita.

- Sparsi per terra ci sono pacchetti vuoti di noccioline e vaccate simili.

In generale un Vero Programmatore puo' lavorare anche 30 o 40 ore di fila, anzi, di solito lavora molto meglio sotto pressione.

Fino a qualche tempo fa si concedeva dei pisolini mentre il computer compilava il programma, ma purtroppo il diffondersi di computer e periferiche veloci ha reso questa pratica difficile.

In generale un Vero Programmatore se ha 5 settimane per terminare un programma passa le prime 4 cincischiando con aspetti secondari, ma interessanti, del progetto, mentre il grosso del lavoro viene fatto in una settimana di lavoro ininterrotto.

Non chiedete mai ad un Vero Programmatore quanto tempo ci voglia per terminare un programma, e se il cliente o il principale intendono ridurre il tempo a disposizione, il Vero Programmatore si preoccupera' della suo programma, pensando ad esso come ad un bambino nato prematuro da sistemare in una culla incubatrice.

Questo provoca sempre grosse preoccupazioni al principale che teme sempre che il lavoro non sia mai pronto in tempo, ed offre al Vero Programmatore una buona scusa per non scrivere la documentazione.

## 7) VARIE ED EVENTUALI

Il Vero Programmatore a volte puo' scordare il nome della moglie, dei figli o della ragazza, ma sa a memoria il codice ASCII.

- Il Vero Programmatore malgrado spenda tutto per tenere in funzione il suo vecchio computer Z80, e abbia a che fare con il fisco solo per la dichiarazione annuale delle tasse, è in grado di elencare a memoria il proprio codice fiscale, anzi se è effettivamente un vero programmatore non lo conosce a memoria ma lo ottiene tramite complicati calcoli a mente.

Il Vero Programmatore non si cura della tastiera, le sue dita si adattano automaticamente a qualunque layout.

Il Vero Programmatore sa che anche avendo 8 mega di RAM questa non sara' mai abbastanza, e percio' tenta di fare programmi piccoli.



Il Vero Programmatore tiene sempre i backup da quando ha dovuto riscrivere 327000 linee di assembler 68020.

Il Vero Programmatore scrive programmi di pubblico dominio, anche se di solito sono programmi talmente specialistici che serviranno solo ad altre tre persone al mondo oltre a lui.

## 8) ALTRI COROLLARI

- Il Vero Programmatore si trovava a suo agio con il Fortran in quanto consentiva la programmazione a spaghetti senza limitazioni.

- Va comunque detto che il Vero Programmatore e' in grado di scrivere programmi a spaghetti in qualsiasi linguaggio. In questo senso, il C va a pennello per la sua capacita' di scrivere programmi Write-only che nessuno, a parte un altro Vero Programmatore, sara' mai in grado di decodificare.

- Il Vero Programmatore non mette mai commenti perche' a suo parere il codice e' autodocumentante. Questo vale anche per i dump esadecimali di codice assembly.

- Nel tempo libero, il Vero Programmatore va abbastanza spesso in discoteca, ma si limita ad osservare il gioco di luci. Ultimamente, viene stranamente attratto dal terminale del controllore laser.

- Ai funerali di un collega, il Vero Programmatore commenta: "Peccato.. la sua routine di sort  $O(\log N)$  stava quasi per funzionare"

- Le Vere Programmatrici esistono in ragione di 1 per ogni 256 Veri Programmatori, come tale la probabilita' di incontrarne una e' estremamente bassa.

- Il Vero Programmatore ha scarsa considerazione degli utenti, ritenuti ad un livello troppo basso. La probabilita' di trovare un utente competente e' stimata inferiore a quella di trovare una Vera Programmatrice.

- Il Vero Programmatore conta esclusivamente in base due.

## CONCLUSIONI:

- Il Vero Programmatore edita direttamente il file Postscript di un documento, se deve modificarlo.

- il Vero Programmatore conosce sempre almeno 16 cifre di  $\pi$  greco, di cui conosce anche la rappresentazione IEEE in esadecimale, e (se anche fisico) tutte le cifre di  $e$  (e' definito con 9 cifre), in modo da non aver bisogno di noiosi include files.



- Il Vero Programmatore e la programmazione ad oggetti: se costretto a simili pratiche, il Vero Programmatore PRIMA scrive il programma, e POI, quando funziona, ne fa un'analisi ad oggetti. Per nessuna ragione comunque modifichera' il codice gia' scritto per conformarlo all'analisi.

Comunque inserira' nel programma un numero sufficiente di variabili globali usate da TUTTE le classi, in modo da renderne impossibile la manutenzione da un mangiatore di Quiche (vedi paragrafo sulla sicurezza dei programmi).

Il Vero Programmatore chiama le variabili con nomi autoesplicativi di massimo 5 lettere (es. CVfrZ). Solo mangiatori di Quiche usano nomi tipo "Massimo\_Numero\_Di\_Dipendenti" per una variabile. Se un Vero Programmatore usa un nome simile, probabilmente la variabile indica la velocita' terminale di uno ione in una nube molecolare (il codice e' stato riciclato efficientemente da un programma di contabilita').

Nel caso in cui una persona con il DNA da Vero Programmatore lavori in COBOL, e per di più sia obbligato a fare programmi gestionali, magari in una compagnia di assicurazione, anche continuando ad evitare di scrivere commenti e documentazione, sarà perennemente depresso, la sua vita sociale ne risentirà (il Vero Programmatore non ha vita sociale, quello che considera socializzare è scrivere istruzioni in codice macchina) e se qualcuno gli chiede quale sia il suo lavoro risponderà semplicemente che è un impiegato.



## GLOSSARIO

(in fase di aggiornamento)

### *Allocazione dinamica della memoria*

Associazione tra specifiche locazioni di memoria e le componenti individuali della variabile in fase di run-time.

### *Assegnamento*

Istruzione elementare non strutturata

### *Espressione*

Consiste in un termine, seguito da un operatore, seguito da un termine (i due termini costituiscono gli operandi dell'operatore).

### *Parola*

Dimensione minima di memoria allocabile, espressa in bit o in byte

### *Run-time*

Momento in cui viene eseguito un programma

### *Termine*

Può essere o una variabile, rappresentata mediante un identificatore, o un'espressione racchiusa tra parentesi



## RINGRAZIAMENTI

Con la presente nota voglio ringraziare tutti coloro che utilizzano questa documentazione, che segnalano imperfezioni od errori o che inviano consigli per modificarla<sup>30</sup>.

Grazie per il vostro aiuto.

---

<sup>30</sup> Per l'invio di segnalazioni inerenti a questa documentazione scrivere all'indirizzo mail [g.iozzelli@tidestudio.com](mailto:g.iozzelli@tidestudio.com). Grazie.



## BIBLIOGRAFIA

(in fase di aggiornamento)

O.J. Dahl, E.W. Dijkstra e C.A.R. Hoare, *Structured Programming* (New York: Accademy Press, 1972)

C.A.R. Hoare, *Notes on Data Structuring*

R. Bayer e E. McCreight, *Binary B-trees for Virtual Memory*, Proc. 1971 ACM SIGFIDET Workshop, San Diego, Nov. 1971

Niklaus Wirth, *Algorithms + data structures = programs*, 1976 Prentice-Hall inc., Englewood Cliffs, New Jersey

Alessandro Bellini, Andrea Guidi, *Linguaggio C – guida alla programmazione*, 1999 McGraw Hill Libri Italia srl.



***Pagina bianca preposta a contenere annotazioni.***



***Pagina bianca preposta a contenere annotazioni.***