NUMERO: 1736A - ANNO: 2015

A P P U N T I

STUDENTE: Lurgo Chiara

MATERIA: Informatica - prof. Rebaudengo

04-03-2013

* INFORMATICA: scienza che rappresenta e manipola le informazioni
  → elaborazione di dati in ingresso (calcolo) e rappresentazione in output (uscita)
  → ci sono diverse tipologie di dati in ingresso (numeri, lettere, immagini) ma i calcolatori possono lavorare solo con numeri binari → si passa da un dato nel suo formato al formato del calcolatore

• PERVASIVO: è in mezzo a noi
→ tutti i dati vengono trasformati in numeri binari (rappresentato solo da 0 e 1)

BIT: cifra di un numero binario
→ un numero binario è composto da una serie di bit
BYTE: pacchetto di BIT → 8 BIT → unità della misura della capacità di memoria
WORD: 2 byte o 4 byte

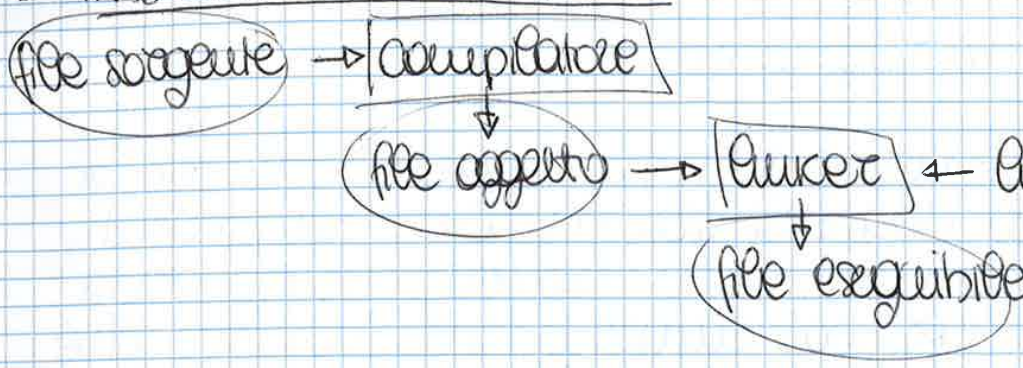
→ 2 tipi di elaboratori: - uso generale → + app contemporanee
- dedicati → dedicati ad una specifica applicazione

• calcolatore: hardware: parte fissa
software: programmi

CPU + memoria + programma = computer

* Elaboratori (general purpose): - personal (client)
- workstation
- server
- main-frame (host)

TRADUZIONE DI UN PROGRAMMA



- FILE: unità di memoria che contiene una serie di dati
  ↳ memorizzata su disco.
  I dati possono essere di vari generi
  → anche i programmi sono file

→ FILE SORGENTE: file scritti dal programmatore che sono x generare un programma. Possono anche essere scritti in diversi linguaggi. Questo codice deve essere tradotto in programma
(codice ")
↓
scritto con i linguaggi di programmazione (x es C, Java, ...)

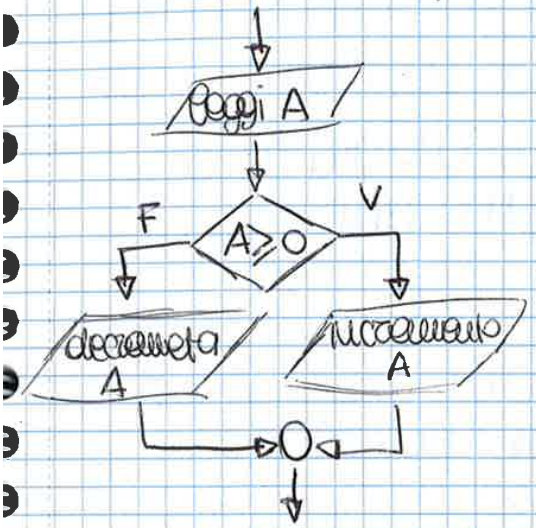
→ ogni file sorgente genera un file oggetto che xo' devono essere messi insieme x generare un unico file eseguibile (programma)

→ librerie: programmi già pronti messi a disposizione dal programmatore

- tutti i programmi sono salvati su disco (unità di memoria permanenti) ma esistono
  - PROCESSORE
  - MEMORIA
  - INPUT/OUTPUT

→ il programma deve risiedere nella memoria principale. La CPU x eseguire un programma deve prendere il programma dal disco e trasferirlo nella memoria principale

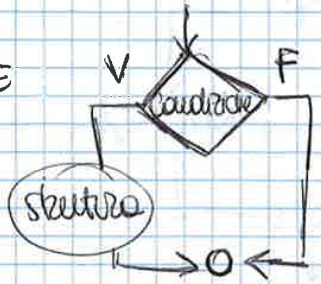
→ BLOCCO DI DECISIONE con un ingresso e due uscite con in mezzo una condizione che è un'espressione che da solo 2 possibili soluzioni (o vera o falsa) → di tipo booleano (x es. $x>2$) → si può quindi continuare dal ramo vero o dal quello falso



← si può procedere o in un modo o in un altro (sono di alcuna escludenti)

→ mettendo insieme i vari blocchi elementari si costruisce un ciclo → alla fine devo arrivare ad un unico flusso che può essere strutturato o no (non so è se è XK ci sono salti incondizionati)

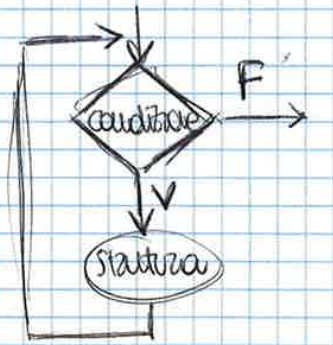
→ IF-THEN-ELSE



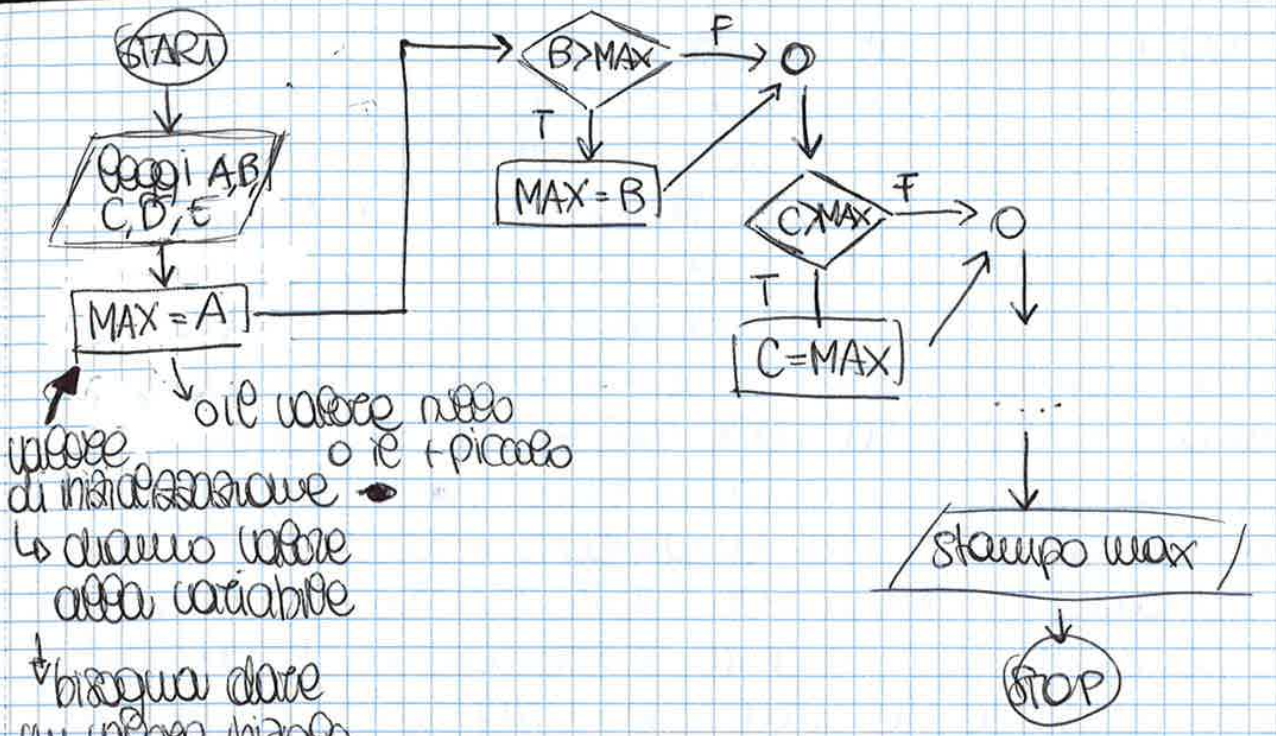
se, come in questo caso, c'è un'azione solo in un caso (qui V), nel caso contrario passa direttamente all'operazione seguente

→ CICLO (si ripete + volte la stessa azione)

- WHILE-DO



→ se il programma non ha una soluzione si ha un ciclo infinito



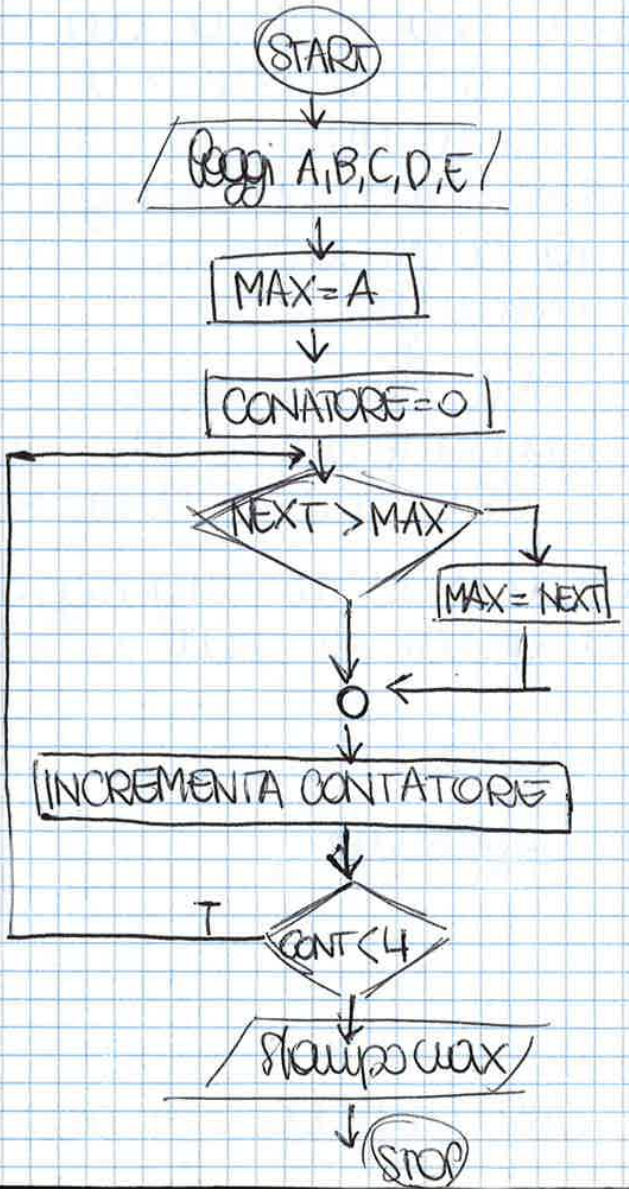
valore di inizializzazione

↳ diamo valore alla variabile

↓ bisogna dare un valore iniziale ad una variabile

oppure: il valore più grande o il più piccolo

→ oppure:



① KEYWORDS: ogni linguaggio ha parole riservate (hanno uno specifico significato per quel linguaggio) → si usano x fare determinate azioni e non possono essere usate x altri scopi. Sono i "mattoni" della sintassi del linguaggio

② DATI: bisogna lavorare con dati input, intermedi e fornire dei dati output. Sono memorizzati in qualche unità di memoria (free). Possono anche essere presi da una memoria secondaria ma x essere usati devono essere portati in quella primaria. Sono rappresentati da BIT. Esistono varie rappresentazioni che ci permettono di capire che cosa rappresenta un dato (fondamentale)

③ IDENTIFICATORE: modo x dare un nome ad una variabile (x es. x) → sono simboli che il programmatore dà a vari oggetti (x es. variabili, funzioni, ...) → tutto ciò che ha un nome di fantasia è un identificatore.

Ci sono diverse rappresentazioni di dati che possono occupare + o meno memoria → x accedere ad ogni spazio di questa memoria bisogna dare un nome a questa zona di memoria che contiene un dato.
Ogni dato ha un nome e un TIPO (x es. numero intero, carattere, ...) → specifica la rappresentazione del dato.
Ogni dato ha una sua rappresentazione → occupa un certo n° di byte

MODALITÀ DI ACCESSO: 1-costanti
2-variabili
→ una costante ha un valore iniziale che rimane uguale fino alla fine (viene solo letta) mentre una variabile può essere modificata durante l'esecuzione del programma (può essere [illegible] una volta che serve) → può essere modificata

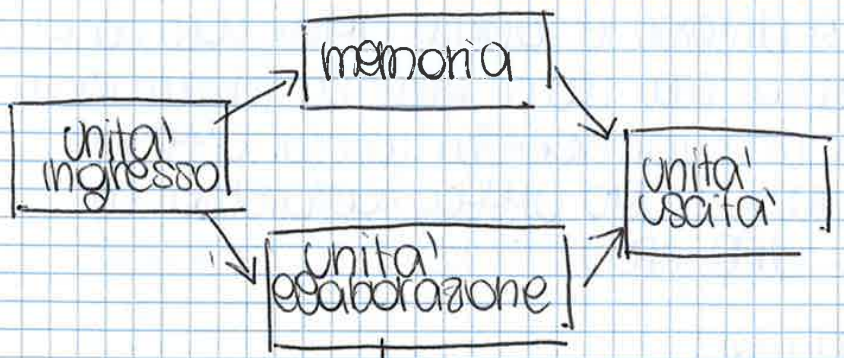
→ La memoria è divisa in siti, ognuno con un proprio indirizzo specifico → il compilatore deve associare ad ogni variabile un indirizzo.
→ ogni variabile ha un indirizzo x fare in modo che venga detta dalla CPU che andrà a leggere tanti byte quanti il tipo indica che il dato va occupa

④ ISTRUZIONI: operazioni che sistema di eseguire a quella macchina

1- PSEUDO-ISTRUZIONI: sono istruzioni che il programmatore scrive e vengono interpretate dal ~~calcolatore~~ compilatore ma non sono eseguibili → alla fine non vanno ad eseguire su quella macchina

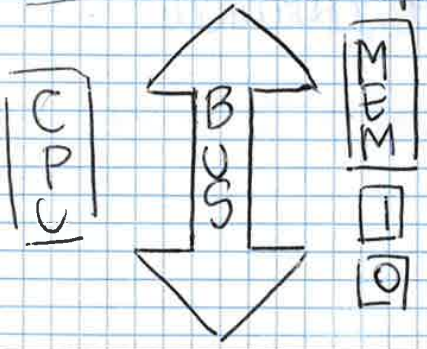
2- ISTRUZIONI ELEMENTARI: operazioni svolte dall'hardware.
3- " DI CONTROLLO DEL FLUSSO: (x es if then else) → va a dirigere il flusso in un punto o in un altro del codice in base al risultato di un'operazione → vanno a modificare l'ordine normale del flusso

→ ARCHITETTURA DEGLI ELABORATORI



→ schema base calcolatore: dispositivo che attraverso lo svolgimento di un programma elabora dei dati input e mostra dati output

(parte che svolge le operazioni)



→ STRUTTURA DI VON NEUMANN

* **BUS**: canale di comunicazione che permette alla CPU di comunicare con gli altri dispositivi.
  Trasmette un dato alla volta
  - **frequenza**: velocità con cui i dati possono essere trasferiti
  - **ampiezza**: n° di bit di cui è costituito un singolo dato
  - → un blocco di bit può essere trasferito in parallelo (tutti insieme) o in modo seriale (non tutti insieme)
  - → se mal dimensionato può essere un collo di bottiglia (ciò che determina il limite delle prestazioni → tutto viene rallentato da questo problema

→ BUS: 1- DATI (D BUS)
2- INDIRIZZI (A BUS)
3- CONTROLLO (C BUS)

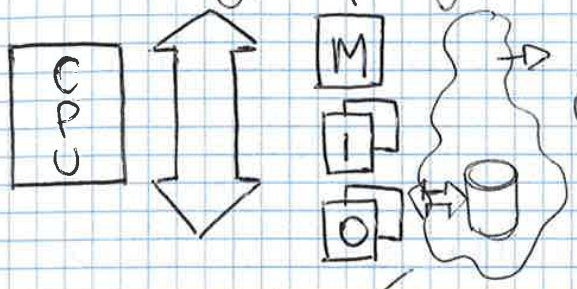
1- bidirezionale x dati, in ingresso x input, in uscita per output
2- in uscita x tutti
3- bidirezionale " "

1- trasferisce i dati ed è bidirezionale xk possono essere sia scritti che letti
2- solo in uscita xk la CPU specifica l'indirizzo → indica qual è la direzione (sempre in uscita)
↳ contiene gli indirizzi → in base alla sua grandezza posso dire quanta memoria è accessibile. Su A BUS viaggiano numeri binari → il numero possibile di indirizzi è $2^{|\text{dimensione di A BUS}|}$ (può andare da tutti 0 a tutti 1)
→ max memoria $= 2^{|ABUS|} \times |DBUS|$ bit
↓ n° parole ↓ grandezza parola

• es → $2^{20} \times 2\,byte = 2MB$

→ oltre alla memoria centrale RAM esistono memorie esterne (di massa → dischi)

[RAM = random access memory → memoria ad accesso casuale → il dato viene preso casualmente e non bisogna fare tutto il disco]



→ max quantità di memoria esterna dipende dal bus di I/O (quello a cui sono collegate le periferiche)

mondo = mondo esterno = tutti i dispositivi periferici

unica parte visibile dalla CPU → x questo il collegamento, è la parte nella quale la CPU scrive e legge i dati

→ la CPU deve programmare (configurare) questi dispositivi
↳ il sistema operativo va a scrivere le parole opportune x configurare all'interno di alcuni registri interni che hanno un loro indirizzo (e come se fosse delle locazioni di memoria). Non sono xò viste (4 o 6)

→ le unità I/O devono trasferire le informazioni dal mondo umano a quello digitale

*BUS ESTERNI: canali di comunicazione esterni oltre ai bus locali che servono x le operazioni I/O

*REGISTRI: sono una parte di hardware che serve x memorizzare dati nella CPU
→ se le variabili sono nei registri interni sono immediatamente disponibili (sono al top della gerarchia di

* clock: segnale di temporizzazione che genera un segnale logico con una determinata

→ maggiore è la frequenza e maggiore è il n° di azioni eseguite → + è veloce e il computer

→ il duty cycle è l'intervallo di tempo in cui viene solo un'operazione elementare ed è un multiplo intero del periodo di clock

→ l'esecuzione di un'istruzione richiede un numero di cicli macchina, variabile a seconda dell'istruzione

---

18-03-2013

LINGUAGGIO C

- imperativo: ho comandi con le istruzioni
- ad alto livello: un po' astratto
- strutturato: ha una rigorosa sintassi con cicli
- tipizzato: dobbiamo specificare il tipo di ogni oggetto
- semplice: ha solo 30 parole chiave
- case sensitive: se una parola è scritta in maiuscolo o minuscolo cambia significato
- portabile: posso programmare su windows e portarlo poi su linux o mac, quello che cambia è il compilatore (non dipende dal linguaggio)
- standardizzato: dall'anno che non ha rilasciato la standardizzazione c89 che è poi stato poi standardizzato in c99 dall'iso → siamo fino al C11

→ linguaggi + usati sono C, C++, Java, C#
→ il C è x6 la base di tutti i linguaggi successivi

→ commenti: vengono usati dal programmatore x far capire a chi legge a cosa si riferisce una variabile o un'altro

→ viene tutto degli mai ma ignorato dal computer

→ si scrive /* ~~~ ~~ ~~~ */ (la parte centrale viene ignorata dal computer)

→ + commenti possono essere annidati, quindi non posso scrivere un commento dentro un commento (/* ~~~ /* ~ . ~ */ ~~ ~ */)

→ commento di riga: invece di chiudere il commento (usiamo // e il commento terminerà appena vado a capo → è + veloce xk scrivo meno e non ha fine xò è stato introdotto nel C99 quindi le C98 seguono ancora

→ parole chiave: 32 → non possono essere usate come variabili. 32 corrisponde a 2^5 → occupano solo 5 BIT. Essendo così poche xò alcune sono usate con significati diversi in contesti diversi

*STRUTTURA GENERALE PROGRAMMA C

→ possiamo usare gli oggetti creati al di fuori di fuori

parte dichiarativa globale

main ()

{

parte dichiarativa locale

parte esecutiva

}

può anche essere scritto int main (void) int main () main (void)

→ posso usare solo le queste parole

→ nella parte dichiarativa globale devo andare a specificare il tipo degli oggetti

→ il main è racchiuso tra le graffe

→ parte dichiarativa locale: = alla globale,

specifichiamo subito il compilatore di vedere come variabile

→ i dati vanno dichiarati all'inizio, in questo modo se sbaglio a scrivere la variabile verrà segnalata come errore e non considerato come nuova variabile

→ ci sono 4 tipi (sono 4 parole chiave)
- **char** (tipo + piccolo di C, 8 BIT, contiene caratteri ASCII (tabella che contiene tutti i simboli presenti nella tastiera) → 256 simboli) occupano 8 BIT xk è 2^8 → è l'unico di cui conosco la grandezza mentre x gli altri non lo so xk nessuno standard lo specifica
- **int** (numeri interi)
- **float** (" reali con singola precisione)
- **double** (" " " doppia ")
  ↳ anche + numeri dopo la virgola ma occupa + memoria

→ si possono anche specificare altre caratteristiche:

• segno: signed / unsigned
(applicabili a char e int)
- signed: valore numerico con segno
- unsigned: " " senza "
  ↳ se x es so che i miei valori sono sempre positivi posso scriverlo, in questo modo uso il segno e ho + spazio x il numero + aggiungo il compito del compilatore

Se sono sotto int il compilatore presume che sia signed

• dimensione: short / long
(applicabile a int)
→ con long int posso salvare numeri + grandi.
→ se sono sotto int do solo il compilatore presume che sia short

*COSTANTI: valore fisso x tutto il programma → dati non modificabili. Devo dargli associargli un nome e un tipo

→ const <tipo> <costante> = <valore>

↳ parola chiave → se non lo specifico è come ~~cost~~ variabile

→ CONVENZIONE: identificatori delle costanti in MAIUSCOLO

- esempio: const double PIGRECO = 3,14159;
  const char SEPARATORE = '$';
  const float ALIQUOTA = 0,2;

- esempi → char 'f'
  → int, long, short 26 / 0X1a ← codifica esadecimale (in base 16) / 26L
  → float, double -212,6

→ COSTANTI SPECIALI: caratteri ASCII non stampabili e/o "speciali"
esempio '\n' → x andare a capo

*VISIBILITA' DELLE VARIABILI (in generale dei dati)

Ogni variabile ha un suo spazio dedicato in memoria

- variabili globali: hanno una visibilità globale → possono essere lette e modificate da tutto il programma
  ↳ definite all'esterno del (main())

- // locali: sono visibili solo all'interno della funzione a loro adibita → quando la funzione finisce la variabile viene eliminata dalla memoria
  → una variabile locale del main viene vista solo dal main
  ↳ definite all'interno del main()

→ le potenze di 2 si ripetono di 10 in 10

$2^{10} = 1024$ K

1024 × 1024 = M (mega)

1024 × 1024 × 1024 = G (giga)

(tera)

(peta)

...

$2^0 = 1$	$2^5 = 32$
$2^1 = 2$	$2^6 = 64$
$2^2 = 4$	$2^7 = 128$
$2^3 = 8$	$2^8 = 256$
$2^4 = 16$	$2^9 = 512$

→ conversione $1101_2 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 8 + 4 + 1$

da binario a decimale → $= 13_{10}$

→ da decimale a binario → es:

27	13	6	3	1	0
1	1	0	1	1	

diviso × 2 — quoziente

resto

→ ottengo un numero binario invertito

1 1 0 1 1

→ 1 BYTE = 8 BIT

1 WORD = 2 BYTE = 16 BIT

1 1 0 1 1

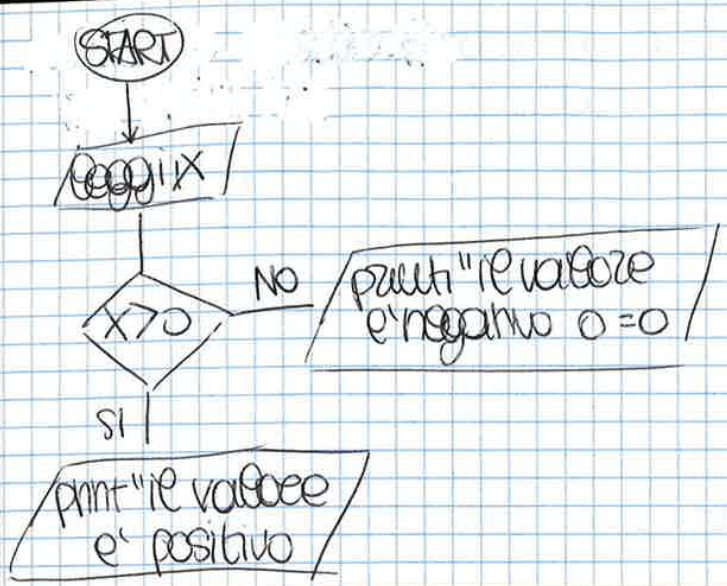
↓ Most Significant Bit

↓ LSB

*ERRORI: il computer non può eseguire il programma

*WARNING: imprecisione con cui il programma viene dato da eseguire

→ se modifico il testo sto modificando la sorgente e non l'eseguibile → devo di nuovo fare build



21-03-2013

- 1 bit ~ 2 numeri
- 2 bit ~ 4 "

→ intervallo di valori → n° in base 10 $0 < x$

→ somma in binario

regole base: 0+0 = 0
0+1 = 1
1+0 = 1
1+1 = 0 (con riporto 1)

xk sarebbe 2 ma 2 è la base

```
1 1 1 1
  0 1 0 1 1 +
  1 0 1 1 0 =
-------------
1 0 0 0 0 1
```

→ in questo caso abbiamo una parola che occupa un bit in + in fatti il 1° corrisponde a 11, il secondo 22 e il risultato 33

→ sottrazione

0-0=0
0-1=1 (prestito 1)
1-0=1
1-1=0

```
  1 1
  1 0 0 1
  0 1 1 0
---------
  0 0 1 1
```

prestito da

```
1 1 1 1
1 0 0 0 0
0 1 1 0 1
---------
0 0 0 1 1
```

è come se facessi 2-1

3-1

* SISTEMA ESADECIMALE

base 16

- cifre: {0, 1, ..., 9, A, B, C, D, E, F}

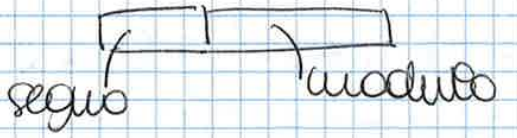
→ conversione 10111001 ← raggruppo a 4

B 9

* NUMERI CON SEGNO (RELATIVI)

Ha: 1 bit dedicato al segno (solitamente MSB)

- 0 (+)
- 1 (−)



- esempio:

(0)	001100	+12
(1)	10101	−21

$+3_{10} \rightarrow (0)011_{M\&S}$

$-3_{10} \rightarrow 1011_{M\&S}$

→ svantaggi: → il numero zero può essere rappresentato con +0 e −0

0000 → +0

1000 → −0

→ se facciamo x OR una somma tra due numeri con segno se sono tutti e 2 positivi il risultato è la somma dei moduli con segno +, se sono 2 negativi è la somma dei moduli negativa ma se sono 1 positivo e uno negativo bisogna fare il 1° − il modulo del secondo se $B<0 \Rightarrow A-|B|$

$A>0$

RAPPRESENTAZIONE NUMERI REALI

25-03-2013

* rappresentazione in virgola fissa

2 1 0	-1 -2 -3
10110	10100

← in binario

• ← virgola

2^1 2^{-2}

→ virgola fissa: dedico una parte alla parte intera (intera) e una alla parte frazionaria (frazionaria) e un'altra a quella dopo la virgola → vengono meglio specificati dei bit → si passa alla rappresentazione in virgola mobile

* mobile

$3,14 = 0,314 \times 10^1$

$0,0001 = 0,1 \times 10^{-3}$

$137 = 0,137 \times 10^3$

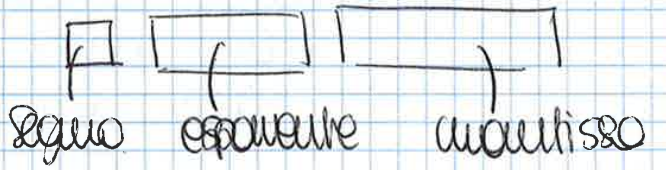
→ è sempre possibile trasformare un numero reale in un modo (formato) standard

$\Rightarrow N = \pm M \times 2^E$

M = mantissa

E = esponente

⇒ RAPPRESENTAZIONE:



* 2 possibili formati:

① IEEE 754 SP (precisione singola)

segno	esponente	mantissa
1 bit	8	23

② IEEE 754 DP (precisione doppia)

segno	esponente	mantissa
1	11	52

10110 . 10100 → 1.011010100

+4 →

- M = 011010100
- E = 4
- S = +

*ELABORAZIONE INFORMAZIONI NON NUMERICHE

→ se la quantità fisica, si può associare ad esso un codice numerico (binario) → si codifica con una parola binaria con n bit.

→ dati N bit si possono codificare 2^N oggetti distinti

(se ho 10 oggetti ho bisogno di 4 bit → $2^4 = 16$)

$2^3 = 8$ → non sufficienti

*CODICE ASCII

È fissa e occupa 8 bit, sono numeri, lettere + simboli

→ ad ogni carattere è associato un codice ascii

- es: A = 01000001
- es: x fare C maiuscola = C+ 'A' − 'a'

distanza tra A e a

→ ogni riga termina con un simbolo ('a capo') →

→ terminatore di riga

- WINDOWS = CR+LF
- MACOS = CR

→ TESTO: - formattato → memorizzato in sequenze di byte che definiscono l'aspetto del testo

- non formattato → sono memorizzati unicamente i caratteri che compongono il testo

- es:
```
int x,y;
int var;
x=10;
y=20;
var=max(10,20);
printf("Il massimo e': %d", var);
```
↳ deve stampare il "testo" e poi la variabile: associa ad ogni % la variabile corrispondente

accesso solo a 1 %d

in questo caso solo 1 variabile

→ es: printf("Il massimo e': %d %d", 2*var, 10+2);

*DIRETTIVE: %d → intero
%u → unsigned
%c → carattere
...

- es:
```
int x,y;
int var;
char car='a';
x=10;
y=20;
printf("car=(%c) codice %d \n", car, car);
```
→ inserisco 1 carattere e voglio visualizzare un carattere e il codice corrispondente

car=a, codice 97 (corrispondente n° in ascii)

---

*scanf()

→ Questo svolge una differenza fondamentale: sono tutte e due funzioni → contengono parametri in numero variabile (uno oppure 1).

→ quando arriva una scanf viene ricevuto un dato dalla tastiera → il programma si interrompe XK è in attesa di un dato

- scanf:

[*] → serve x saltare determinati caratteri che in realtà c'erano: x es non voglio dare valore alle cifre dopo la virgola ⇒ scrivo

"%*d %*d %d %d", &x, &y

*PRE-PROCESSORE C

- direttiva #include serve x inserire un file - viene usata x aprire librerie dove sono conservati dati forniti dal compilatore → alle fine non faccio tutte ma solo quelle che mi servono → @ lui posso quindi includere solo le funzioni che mi servono.

→ si può anche includere un file inserito dal programmatore

- es: #include "file.h" ① va a sostituire questo rigo e lo sostituisce con il file

② continua ricercando nel file

quindi x es posso salvare in un file tutte le variabili globali e poi inserirle così → evito di scrivere + volte

→ si può usare per: ① individuare e includere una libreria

② scrivere un pezzo del mio codice in un altro file

- direttiva #define: potrebbe servirmi utilizzare dei simboli che vanno a sostituire qualcos'altro → uso un valore simbolico al quale attribuisco 1 valore

- es: N = 10

→ #define N 10
#define M 5

→ 3*5 = 15

#define 3*M

- es: x= 2*3
  x= 2*y ↑ data dal compilatore
  x= 2*y → fatte run-time dopo che x è valore di y ma è una costante una volta compilato non cambia più

operatori di confronto in C → guarda slide

- es a>b — SI
  \ NO

04-04-13

- BOOLEANO: può avere solo due valori: vero o falso
  → sempre confronti tra a e b
- i confronti sono 6 (2 uguaglianza e 4 confronto)
- x C a=b è un'istruzione quindi x prima cosa va a fare un'assegnazione:
  a=4, b=5 → a=b → a=5, b=5
- x C è vero tutto ciò che è diverso da 0 xaio:
  if (a=b) è vero
  → se a=4, b=5 → a=∅, b=∅ → if (a=b) falso

operatori di incremento

++ --

→ incrementano di un'unità

es: X++; = X=X+1;

LOGICA BOOLEANA

Si basa sulla variabile booleana: può avere solo 2 valori (vero o falso)

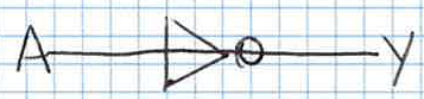
- operatori unari (ad 1 sola operazione) es. NOT
- " binari (a 2 operazioni) es AND
- tabella della verità: comprende tutte le possibilità

A	B	A oper. B
f	f	f
v	f	f
f	v	f
v	v	v

← X AND (guarda sulle)!!!

- operatore NOT: $\bar{A}$ (A negato)

→ La logica booleana definisce gli operatori una circuitalmente vene eseguita l'inversione



$Y = \bar{A}$

→ operatore XOR

A	B	A⊕B
f	f	f
v	f	v
f	v	v
v	v	f

→ vero se sono ≠
falso se sono =

← lo stesso e calcolo i 2 valori parziali

$Y = A \oplus B = A \times \bar{B} + \bar{A} \times B$ →

→ dal problema al circuito

Si prende la tabella della verità e si considerano i casi in cui $Y=1$ ($y=\bar{A}B + A\bar{B}$) → la somma (somma di prodotti)

→ il teorema di de Morgan traduce questa somma in prodotti: passa da NOT o OR a AND

→ quando si fanno i calcoli bisogna anche memorizzare i dati e x fare ciò si usa un elemento logico speciale (flip-flop) che è in grado di memorizzare solo un bit

→ es4: int A, B;
A = A ⊕ B;
B = A ⊕ B;
A = A ⊕ B;

$A \oplus 0 = A$

$(A \oplus (B \oplus B))$

$(A \oplus B \oplus A) = B$

08-04-2013

Dato un numero binario in complemento a due espresso su tre bit b_0, b_1, b_2, scrivere la tabella della verità per la funzione booleana $f(b_0, b_1, b_2)$ che vale 1 quando il numero di '1' è maggiore del numero di '0'

b_2	b_1	b_0	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

④ Le due espressioni booleane

$\bar{a}b\bar{c} + ab\bar{c} + \bar{a}\bar{b}c + \bar{a}bc + abc + ab\bar{c}$ e $b+c$

sono equivalenti? → sono equivalenti se hanno lo stesso risultato in tutti i casi della tabella

A	B	C	B+C	f
0	0	0	∅	∅
0	0	1	1	1
0	1	1	1	1
0	1	0	1	1
1	0	0	∅	∅
1	0	1	1	1
1	1	1	1	1
1	1	0	1	1

$\bar{a}\bar{b}\bar{c} = 1$ ma non è presente → ∅

$\bar{a}\bar{b}c = 1$ → non deve contare
→ se c'è in f = 1
→ se non c'è = ∅

A	B	C	B+C	$\bar{a}b\bar{c}$	$ab\bar{c}$	$\bar{a}\bar{b}c$	$\bar{a}bc$	abc	$a\bar{b}c$
0	0	0	0						
0	0	1	1			1			
0	1	0	1	1					
0	1	1	1				1		
1	0	0	∅						
1	0	1	1						1
1	1	0	1		1				
1	1	1	1					1	

⑤ Si dimostri, mediante la tavola della verità, se la seguente relazione è vera

A or (A and (not B)) = ((not A) and B)

→ quando a fare un'operazione tra due tipi diversi, il dato con rappresentazione + bassa viene tradotto nella rappresentazione + alta

- es:
```
float num;
int a,b;
num = a\b;
printf("%d %f \n", num, num)
```
1 1.0

es → num = 4\3 ⇒ 4.0\3

↓

→ in questo modo ho un'operazione tra un intero ed un reale ⇒ il nº intero cresce di rango

→ se invece voglio mantenere a e b devo usare il CAST che converte 'al volo' il valore

- es:
```
float num;
int a,b;
((float)a)\b;
```
→ metto la parentesi xk se no vedrebbe prima fatta la divisione e poi traduce il risultato in float

↳ a viene tradotta in numero reale attraverso una variabile temporanea

→ poi vado a lavorare con questa variabile e non con a

→ anche b cresce di rango ma x questo non è necessario il cast → si adegua ad a

* SIZEOF()

→ non è una funzione ma un operatore perché deve linguaggio che può essere inserito in un'espressione

→ inserisco una variabile tra parentesi e sizeof mi restituisce il nº di bit che la var occupa

es: main()
{
int A,B,D;
scanf("%d %d", &A, &B);
if (A>B)
D=A-B;
else
D=B-A;
printf

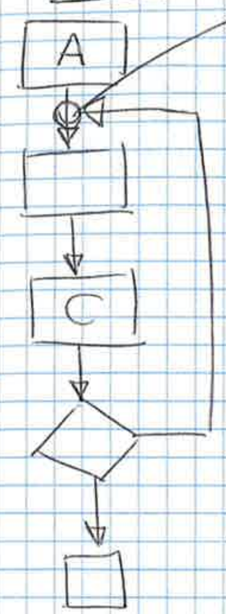
→ vedi slide

* SCELTE ANNIDATE: se x es devono and ma TRUE ho altri
if → IF dentro IF

es: if (C1);
{
A1;
if (C2)
{
A2
}
else
{
A3
}
A4
}
else
{
B
}

devono rappresentare tutti casi diversi

* CICLO



punto di ritorno

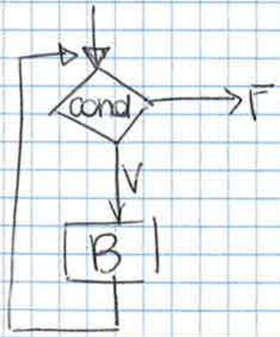
} prima del ciclo

} istruzioni che vengono eseguite

} condizione di verifica (x valutare se rimango o esco dal ciclo

} dopo il ciclo

① WHILE



→ se è F esco dal ciclo se è V vado al punto di ritorno

→ se dopo WHILE ho solo un'istruzione posso non mettere {} se invece ne ho + di una devo x forza

o esempio while scale

N=5
S=∅
i=1

→ facendo una sommatoria potrei andare in overflow ⇒ mi conviene inizializzare S come long int S;

S=1
i=2

S=3
i=3

S=6
i=4

S=10
i=5

S=15
i=6

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 53 di 114

→ ciclo for

11-04-2013

stessa funzione di while ma le parti (inizializzazione, condizione, blocco, incremento) sono tutte raggruppate

- es: voglio visualizzare * 10 volte
  - → ho bisogno di un char (*) e di un int (n° volte)
  - → l'utente deve digitare un int e un char
    - => scanf ("%d %c")
  - → for (i=0; i<N; i++)
    printf ("%c", ch);
    printf ("\n");

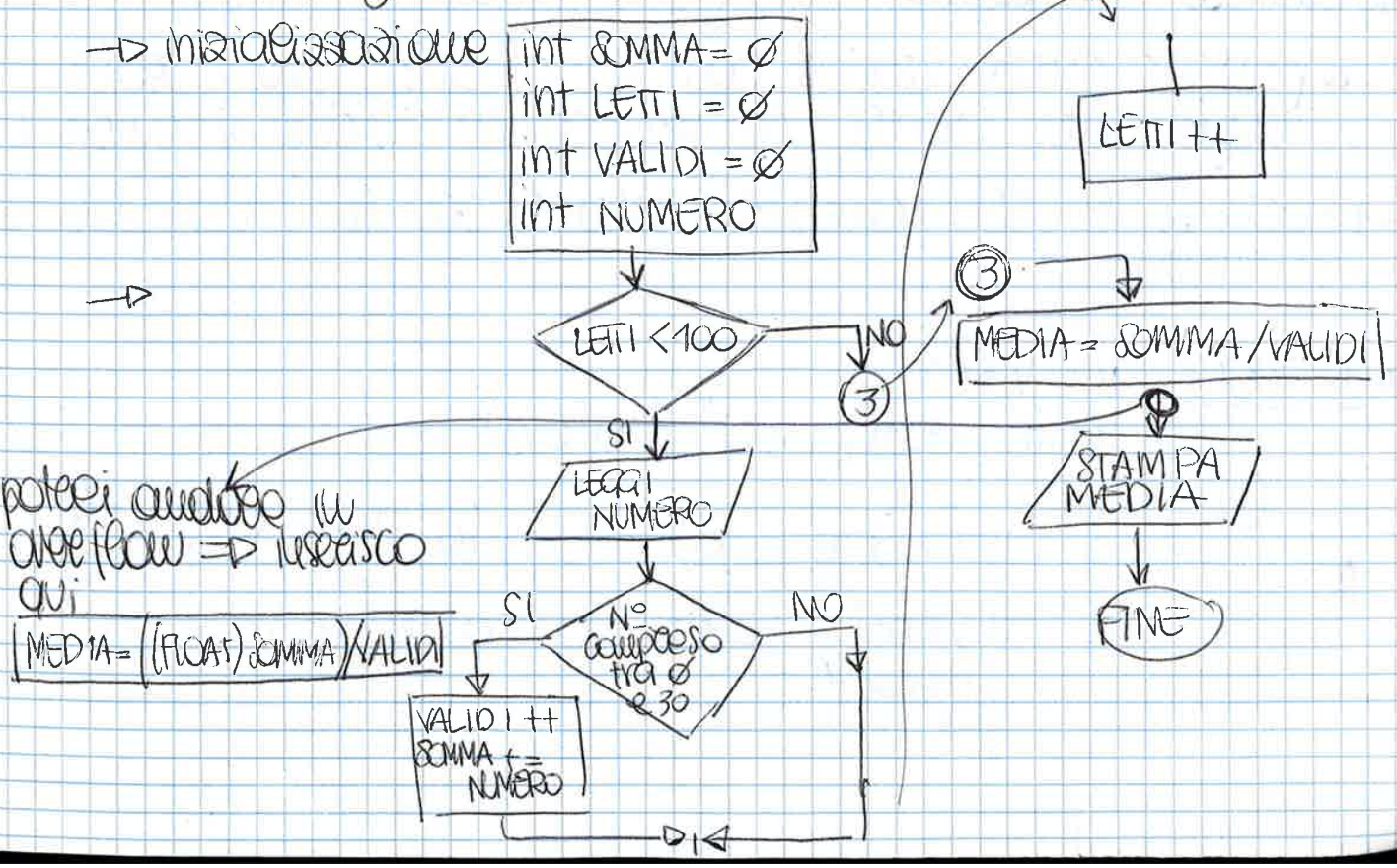
inizializzata a zero in partenza → stampa 1 singolo carattere

↳ eseguo l'ultima volta il ciclo quando i = N-1 ↓
in questo caso so in precedenza quante volte (N) deve essere eseguito il ciclo

- es: Introdurre da tastiera 100 numeri interi, e calcolarne la media. Si controlli che il numero inserito sia compreso tra 0 e 30; in caso contrario, il numero deve essere ignorato

→ inizializzazione

```
int SOMMA = 0
int LETTI = 0
int VALIDI = 0
int NUMERO
```



potrei andare in overflow => inserisco qui

```
MEDIA = ((float) SOMMA) / VALIDI
```

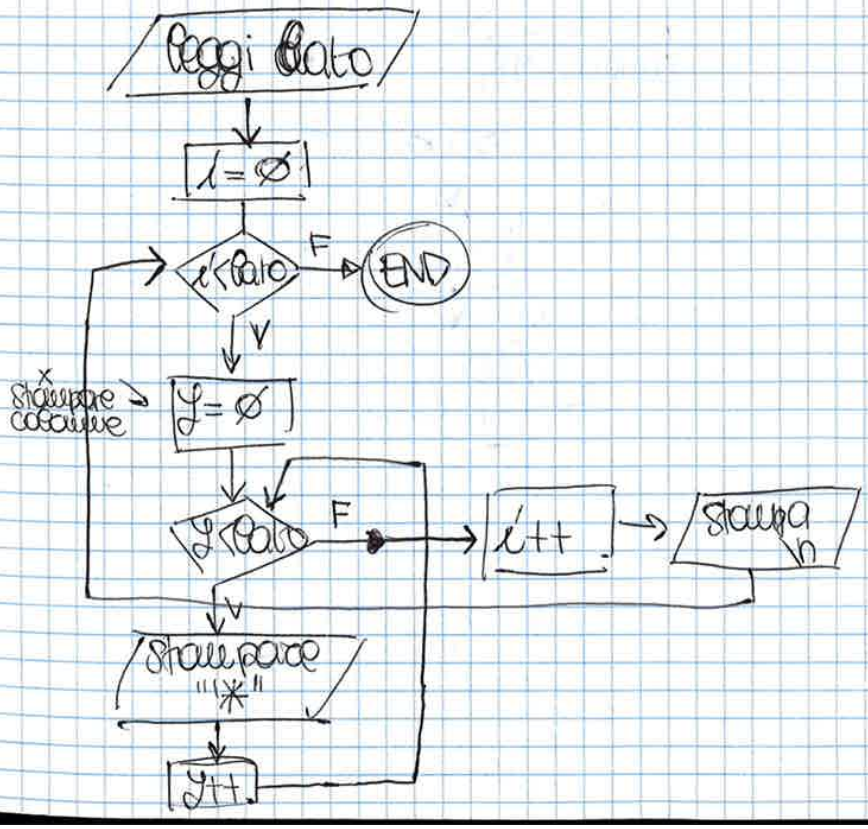
→ x evitare il break — FLAG → (viene usato tipicamente quando posso avere solo 2 valori)

```
int valore, fluito=0;
while (scanf("%d", &valore) && !fluito)
{
  if (valore==0)
  { printf("valore non consentito \n");
    fluito=1;
  }
  else
    {
      ...
    }
...
}
```

→ es ①: uguale uno con CONTINUE → non esce dal ciclo ma salta le istruzioni che lo seguono e salta direttamente alla seconda ?

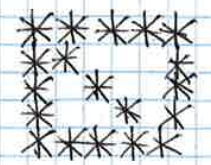
---

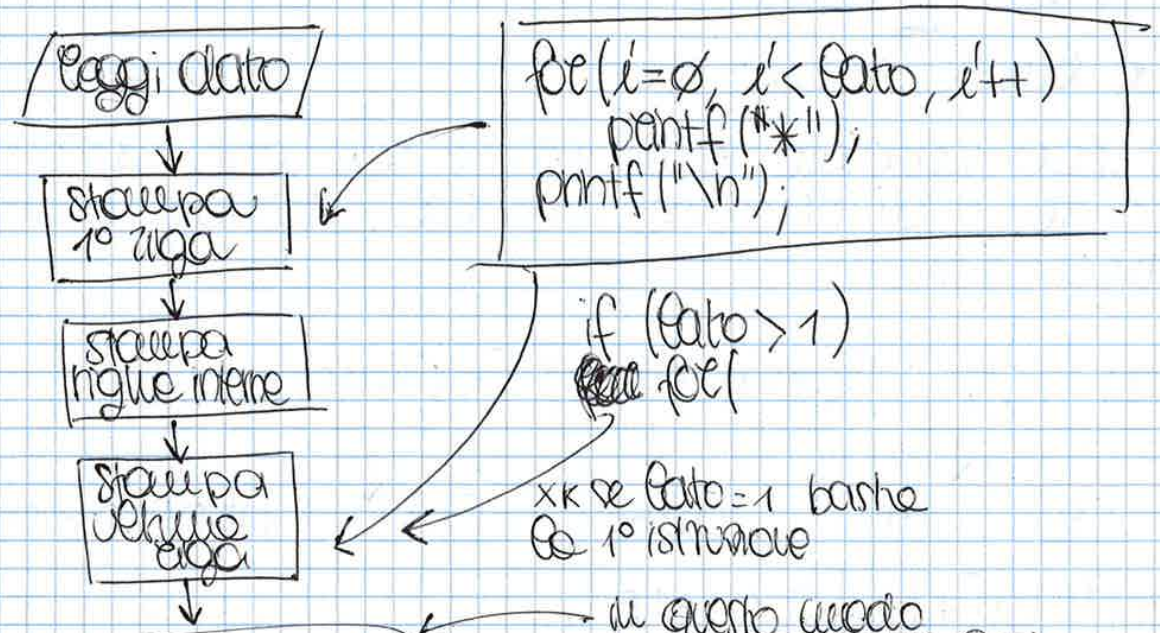
*esercizio: stampare un quadrato usando il carattere * (es: dimensione 4 ****)



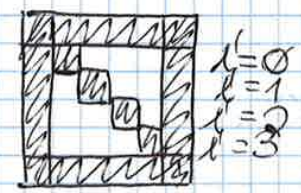
*esercizio: il programma deve stampare il perimetro e la diagonale di un quadrato

→ es dur. 5






```
for (i=0; i<(dato-2); i++)
{
 printf("*");
  for (j=0; j<i; j++)
      printf(" ");
 printf("*");
  for (j=i+1; j<dato-2; j++)
   printf(" ");
 printf("*\n")
}
```



→ no spazi

→ oppure:

```
for (i=0; i<(dato-2); i++)
{for (j=0; j<dato; j++)
   { if (j==0 || (j==dato-1) || (j==i+1)
     printf("*")
     else printf(" ")
 printf("\n)"
```

○ esercizio <1000 18-04-2013

Scrivere un programma che legga un valore decimale e lo converta nella corrispondente codifica binaria

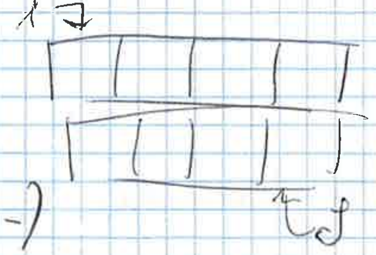
```
                         int i = N-1
if (num >1000)
   printf ("Errore")                 i=0
else {
      while (num != 0) {
      v[i] = (num % 2);
      num = num/2;                   for (i=N-1; i>=0; i--)
      i--;
for (i=0; i<N; i++)
      printf ("%d", v[i]);
printf ("\n")
```

se il num = continua ad essere ≠ 0 continuo a salvare il resto della divisione dentro il vettore poi vado a leggere il vettore con for

→ se voglio

○ esercizio

il 1° elemento del vettore viene copiato in ultima posizione e viceversa
il 2° elemento nella penultima e viceversa
così via



```
int vettore [N];
int vettore_output [N];
for (i=0, j=N-1; i<N, i++, j--)
vettore_output[j] = vettore [i];
```

```
int vettore [N];
int vettore_output [N];
for (i=0, i<N; i++)
vettore_output[N-1-i] = vettore[i]
```

22-04-2013

→ leggo 3 dà 8 → faccio corrispondere l'8 dello 2° al 3 dello 1° e così via

→ così posso avere 2 numeri uguali xk se leggo solo come fare la codifica al contrario

- es:
```
int vettore_criptografia[N]={3,4,0,5,1,6,9,7,2,8};
int pin[M];
int pin_criptografato[M];
int i;

printf("Leggo da tastiera il PIN in chiaro \n");
for(i=0; i<M; i++)
  {
    scanf("%d", &pin[i])
  }
```

x vedere a scrivere pin_criptografato ↓

```
for(i=0; i<M; i++)
  {
    pin_criptografato[i]=vettore_criptografia[pin[i]]
  }

for(i=0; i<M; i++)
  {
    printf("Elemento %d = %d \n", i, pin_criptografato[i])
  }
return 0;
```

cifre ++, → conta fino alla metà il numero di cifre di cui è composto il numero

```
}
[palindromo = 1;]
for(i=0, i<cifre/2, i++)
    if (numero[i] != numero[cifre-1-i]
        {palindromo = 0;
         break;
        }

if (palindromo)
    printf("Il numero è palindromo\n")
else
    printf("  "  "  non "  "  )
}
```

0 1 2 3 4

---

*RICERCA DI UN ELEMENTO

→ dato un valore verificare
- se almeno uno degli elementi del vettore è uguale al valore

V → dove si trova

F → non esiste

→ × faccio introdurre flag

```
int dato;   → dato da ricercare
int trovato; → flag
int pos;    → posizione
```

trovato = 0 → lo inizializzo = 0 → non trovato

```
pos = -1
for (i=0; i<N; i++)
{ if (V[i] == dato)
  { trovato = 1; → se ce ne fosse + di uno mi
    pos = i;       tiene conto dell'ultimo →
  }
```

- es: determinare se una sequenza di dati inseriti è crescente

```
int crescente;
crescente = 1
precedente = INT_MIN;
i=0
{
   if (v[i] < precedente)
      crescente = 0;
   precedente = v(i);
   i=i+1
}
```

```
precedente = v[0];
i=1
```

↳ Faccio già il 1° confronto in questo modo

```
int crescente;
crescente = 1;
i=1
while (i<n)
{
   if (v[i] < v[i-1])
      crescente = 0;
   precedente = dato
```

---

es: conteggiare i dati

```
int conta;
conta=1
i=1;
while (i<n)
{  if (dato[i] != dato[i-1])
      conta = 0;
   i=i+1;
}
```

29-04-2013

*FUNZIONI → funzione ha valore di ritorno, la procedura no

→ ha un valore di ritorno. Se non ritorna ad un valore bisogna anche indicare il tipo

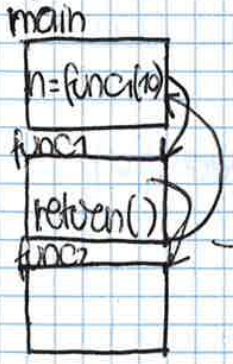
→ un programma ha sempre un MAIN: [redacted] funzione da cui il programma parte → se ho una variabile nel main questa persiste x tutto il programma xk il programma inizia e finisce con il main ma la variabile è anche locale → può essere vista solo dal main stesso

→ es: double func (int x, int y) → numeri non necessari

• PROTOTIPO: stessa della funzione: nome; tipo di valore di ritorno e tipo dei parametri

→ bisogna riportare il prototipo delle funzioni che andrò ad usare, prima del main → richiamo una funzione che andrò ad usare + avanti → in questo modo quando vado a richiamare la funzione nel main, il compilatore sa già cos'è xk è come se l'avessi inizializzata prima del main anche se viene definita dopo → non serve quindi fare se la funzione è definita prima del main e poi usata dopo nel main (non è necessario ma anche consigliato)

→ Nel prototipo serve il nome della funzione, il tipo del parametro di ritorno e anche solo il tipo dei parametri usati (non è necessario il nome)



→ funzioni annidate: una funzione richiama un'altra e così via

→ x e y x es vengono passati a V_1 e V_2; posso modificare V_1 e V_2 ma queste modifiche non hanno mai conseguenze su x e y

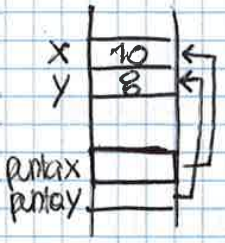
→ i parametri formali invece possono essere modificati: funzionano come variabili locali. La funzione può andare a lavorare con essi

→ è possibile lavorare con gli indirizzi delle immagini

*PASSAGGIO PER INDIRIZZO: il parametro non è + la variabile ma l'indirizzo della variabile stessa → viene passata la posizione della memoria che mi permette di "puntare" la variabile

→ VARIABILE PUNTATORE: variabile che contiene un indirizzo

→ l'indirizzo di una variabile viene copiato in una variabile puntatore



→ in questo modo posso modificare x e y XK conoscendo gli indirizzi posso modificare i valori

[OPERATORE: funzione che lavora con dei dati]

↳ si applica ad una variabile e dà un risultato:

- es &v restituisce l'indirizzo di v

→ non dimenticare di scrivere l'indirizzo vicino &

→ la funzione usa i parametri formali che sono come le sue variabili locali: la variabile puntatore si indica con *

- es: int *a → la variabile si chiama a ed è un puntatore ad intero (è importante specificare il tipo della variabile puntata x indicare lo spazio della memoria

*FUNZIONI LIBRERIA

funzioni predefinite, già fatte e a disposizione, pronte da usare: si usano con #include <math.h>

f. libreria

TABELLA ASCII

02-05-2013

codifica in 7 bit dei caratteri + importanti
→ ci sono anche caratteri non stampabili (caratteri di controllo: primi 31)
→ dal 32 in poi caratteri stampabili
→ ogni carattere viene memorizzato con il corrispettivo carattere ascii
→ una stringa è rappresentata dai caratteri ascii
→ ACQUISIZIONE/STAMPA di un carattere alla volta:
- getchar / putchar → agiscono su un solo carattere

(print) *getchar: legge il valore inserito e lo fornisce come numero corrispondente in ASCII

(scanf) *putchar: trasforma un ascii in un numero corrispondente

→ se scrivo A ① con %c me lo scrive come A
② con %d " " " " numero corrispondente in ascii

① - printf("hai premuto %c\n", tasto)
- printf("Hai premuto ");
putchar(tasto);

→ \0 serve xk mau e' detto due no una stringa
Ea così tutta: x es posso salvare in char saluto[10]

(SI) char s[] = "ciao"

(NO) s = "pippo" ← mau dato eventuale il numero di caratteri perche' prende due solo il numero di celle di "ciao"

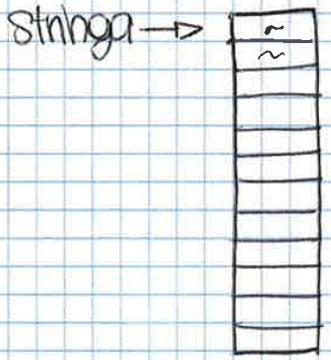
*FUNZIONI SSCANF SPRINTF

→ Ea printf scrive a video mentre Ea sprintf scrive a stringa int stringa[20], a

- es: sprintf(stringa, "%d", a)

→ acquisizione/stampa di una riga alla volta

06-05-2013



char stringa [20]
char * s; s = stringa

e' un puntatore → contiene il numero che rappresenta l'indirizzo della variabile stessa: 1° elemento della stringa

scanf x una riga → char * gets (legge una riga da tastiera fino a \n)

printf x una riga → int puts → stampa la stringa data scelta e aggiunge sempre \n alla stringa

→ se c'è un errore l'indirizzo di s è uguale a 0

- strcat: si usa 2 stringhe s_1 e s_2
  char $s_1[10]$, $s_2[20]$
  → strcat (s_1, s_2);
  concatena il contenuto di s_2 in s_1 una s_2 ha 20 caratteri e s_1 solo 10
  → 

  ← s_2 viene copiato negli ultimi vuoti di s_1, se sono tutti pieni o non bastano, s_2 viene "attaccato" a s_1

- esempio: char str[80]; ← inizializzazione della stringa
  strcpy (str, "these ");
  strcat (str, "strings ");
  strcat (str, "are ");
  strcat (str, "concatenated.");
  puts (str);
  return 0;
  → output: these strings are concatenated.
  → il valore di ritorno di strcat è l'indirizzo della stringa destinazione che non viene modificata

---

- strchr: cerca la 1ª occorrenza del carattere nella stringa → scandisce tutta la stringa x cercare un carattere
  → se il carattere non c'è ritorna un valore nullo
  → se lo trova restituisce l'indirizzo della posizione in cui lo ha trovato ← la dimensione non è esplicitata la troviamo dalla frase +1
  esempio: char str[] = "questa stringa e' un es.";
  char * pch;
  printf ("cerco occorrenza del carattere 's' nella stringa \n");
  pch = strchr (str, 's');

- strcpy: copia (serve x inizializzare)

```
char str1[] = "stringa";
char str2[40];
char str3[40];
strcpy(str2, str1);
```
→ resetta il contenuto precedente e copia quello della seconda stringa dal suo interno

→ si può anche fare
strcpy(str1+1, str2) ← in questo modo il primo resterà invariato quello di str1 e poi copia quello di str2

- strlen → è un int: restituisce la lunghezza della stringa senza includere \0

- strncat: → copio 2 stringhe una dentro l'altra il numero dei caratteri (senza contare \0)

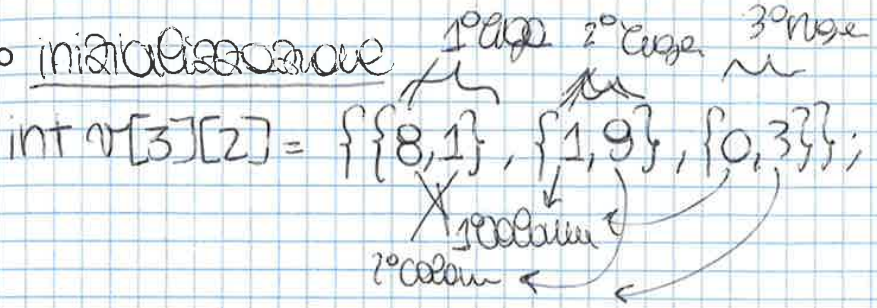
- strncmp: → confronta 2 stringhe solo x il numero di caratteri che voglio

- strncpy: → copia 2 stringhe solo xò x il numero di caratteri che voglio
→ se xò è scritto ha un numero di caratteri minore del numero che ho inserito ⇒ viene copiato una sequenza di zeri che va a completare il numero - gli zeri xò non vengono detti xk sono dopo pieni
→ se invece è + lungo copia il numero di caratteri che gli ho detto ma non include lo \0 che deve andare ad inserire manualmente
- es n = 5
s[5] = \0

- sintassi

<tipo> <nome vettore> [<dim 1>][<dim 2>] ... [<dim N>]

- inizializzazione



- scansione

Servono 2 indici: uno che si muove sulle righe e uno sulle colonne → 1 ciclo for per ogni riga e uno interno x le colonne

↳ stesso caso x la stampa

---

*ARGOMENTI SULLA LINEA DI COMANDO

13-05-2013

Il programma fa parte del software che è una parte del sistema operativo che svolge un programma di funzione. Il sistema operativo gestisce le funzioni attraverso le quali vengono lanciati comandi al sistema, attraverso LINEE DI COMANDO: formate dal <nome> seguito da una serie di possibili funzioni.
C'è uno scambio di comandi tra il programma e il sistema, attraverso dei comandi che vanno ad influire in modo gestito

- esempio: C:\> mypcg <arg 1> <arg 2> ... <arg N>

nome del programma che lo lo scrive → argomenti che vi vengono passati x il programma

- es: C:\> copy file1.txt dest.txt

il 1° è sempre o 1 argomento x

```
int x;
char s[80];
x = atoi(argv[1]);
strcpy(s, argv[2]);
```

→ 1° argomento

→ 2° argomento

→ alcuni argomenti della linea di comando indicano modalità di funzionamento alternative del programma → diversi modi operativi possibili → tipicamente queste alternative due precedute dalle linee

---

*EXIT

→ lui fa uscire in modo drastico dal programma in qualunque posizione si trovi nel codice in cui sono → exit è definita funzione del programma

es: dati numeri interi di N e divisibili per D
N e D letti dalla linea di comando
→ 2 numeri → argc = 3 (nome + 2 numeri)

```
int main(int argc, char* argv[]){
if(argc != 3){
    printf("Numero non valido");
    return 1;
}
if (argv[1] != NULL) N = atoi(argv[1]);   → spezza (ho già fatto prima che è = 0)
if (argv[2] != NULL) D = atoi(argv[2]);
for (i=1; i<=N; i++){
    if ((i%D) == 0){
        printf("%d\n", i);
    }
}
}
```

LAB 8 es3

20-05-2013

cercare 2 occorrenze della stringa con "ch"

```
if (strncmp (s, "ch", 2) == 0
    {
    stringa[i] = 'k';
    for (j = i+1; j < strlen(stringa); j++)
        stringa[j] = stringa[j+1];
    }
else if (stringa[i] == stringa[i+1]
    for (j=i+1; j<(strlen(stringa)); j++)
        stringa[j] = stringa[j+1];
    }
while (gets(stringa) != NULL)
```

quando trova "ch" sostituisco c con k ma poi devo traslare indietro di 1 tutto la stringa

do i+1 poi solo questo

x le doppie

devo scorrere la parola fino a quando non trova EOF

TIPI AGGREGATI

posso aggregare in una nuova variabile tipi diversi

```
struct <identificatore> {
    campi
};
```

nome stesso

"record" → questo aggregato: è come se fosse delle variabili: ogni campo deve avere un proprio nome e tipo

- es: studente: cognome
  nome
  anno nascita

→ posso inizializzare i campi di struct in fase di definizione

→ posso farlo seguendo l'ordine in cui li ho definiti se devo cambiarli all'interno della parte di struct

---

es: typedef struct persona
{
char nome[LEN];
char cognome[LEN];
int età;
float peso; } persona;

```
#define N 4
int matr[N][N];
int main()
{ int i,j;
  int k=1;
  for(i=j=0; j<N; j++)
     matr[i][j] = k++;
  for(i=1, j=N-1; i<N; i++)
     matr[i][j] = k++;
  for(i=N-1, j=N-2; j>=0; j--)
     matr[i][j] = k++;
  for(i=N-2, j=0; i>0; i--)
     matr[i][j] = k++;
  stampa_matrice();
}
```

↑ gira

= void cornice (int inizio, int fine)

→ 1° suddivisione dei file:

BINARI // ASCII

↓

tipo testuale (contengono caratteri ascii)

→ tra quelli ascii ci sono x es anche gli [redacted]. Altri che contengono sia testi che comandi
↳ testo formattato

sono file che contengono valori binari, che non contengono testi ascii → sono l'output di qualunque tipo di programma (file "proprietari" → codifica quello delle XK è di proprietà della produttore)
→ possono anche essere di tipo "standard": in questo caso la codifica è stata resa di pubblico dominio

→ bufferizzare: leggere e salvare il contenuto
→ quando realizzo I/O da file (leggo e scrivo) vado ad usare questa parte di memoria bufferizzata

→ il file di testo viene visto dal computer come un flusso sequenziale di byte → li leggo in sequenza fino all'EOF
→ li leggo in maniera sequenziale quindi impossibile leggere al contrario o saltare un pezzo del file

→ ci sono funzioni che mi permettono di aprire un file e mi danno come risultato un indirizzo
→ devo usare un puntatore x andare a leggere in modo sequenziale un file
→ devo avere puntatori diversi (variabili diverse) x file diversi

① connessione tra variabile e file fisico. Devo usare una specifica funzione:

FILE * fopen (char* <nome file>, char* <modo>)

→ se voglio associarla ad 1 variabile:

f1 = fopen ...

(char* <nome file>) → è una stringa

→ il nome del file posso avere sia solo un nome sia anche il path specifico (es pippo.txt) ma posso scrivere anche il path completo. Se uso solo il 1° xk mi trova il file devo essere sicuro che sia nel folder in cui sto andando ad eseguire il programma se no devo indicare x forza il path

→ il modo: indica come devo aprire il file: 2 modi: lettura o scrittura. Se è lettura il file esiste già, se è in scrittura lo vado a creare o vado a sovrascrivere: se sovrascrivo e quello che scrivo nuovo è di meno di quello che era già scritto non mi mantiene quello che c'era già scritto ma scrive quello nuovo con eof terminando l'accesso

→ ci sono poi anche altre modalità

Quando vado a eseguire la fopen mi può restituire il valore NULL se ci sono stati errori. L'errore lo abbiamo quando il file non esiste. L'errore può esserci anche in scrittura x es se l'operazione di scrittura è impedita all'utente.

Se quando uso fopen ho successo questa mi inizializza la variabile puntatore al file.

→ che andre un valore di ritorno che è NULL se non c'è niente da leggere (fine o quando arrivo a EOF

→ Lettura formattata: int fscanf (FILE* <file>, char* <formato>)

* altre funzioni

① FILE * freopen (const char* filename, const char* mode, FILE * stream)

→ mi permette di riaprire un file e spostarmi in un'altra posizione

↳ file già aperto

② int fflush (FILE* <file>) → va in memoria su disco

→ scarico il buffer di file - se c'è x es prima lo fa automaticamente quindi questo lo faccio raramente x questioni di sicurezza

→ non fa lettura → non sposta il puntatore

③ void rewind (FILE* <file>)

→ torna all'inizio del file

30-05-13

```
FILE *fpin, *fpout;
int x,y;
if((fpin = fopen("estremi.dat","r"))==NULL)
{
  fprintf(stderr,"Errore\n");
  return 1;
}
```

equivalenza a scrivere printf("Errore\n"

stderr file standard che corrisponde ad un file situato nella directory in cui vengono salvati gli errori

03-06-2013

PUNTATORI

bisogna specificare il tipo di variabile a cui il puntatore punta

<tipo> * <identificatore>

XK se gli dico di incrementare devo sapere quanta memoria occupa un dato

- int x
  int y > voglio salvare in px e py rispettivamente l'indirizzo di x e y

⇒ uso & che mi permette di estrarre l'indirizzo

⇒ int x, *px
px = &x

int y, *py
py = &y

printf("%d", x) → 5
printf("%d", px) → 10016
printf("%d", &x) → 10016
printf("%d", *px) → 5

*(&px) → fornisce il contenuto della casella che contiene l'indirizzo di px

→ OPERAZIONI
int *px; → px++, px--

es: int * ptr; int i1, i2, i3 ... ;
ptr = &i1;
ptr++;

↓ mi salta gli indirizzi uno dopo l'altro ⇒ se uso posso scrivere i1, i2 ecc... una x volta accedendo a ptr

→ x trovare caratteri in stringa: e restituisce l'indirizzo

char * strchr (char *s, int c);

→ char * p, *stringa = "stringa prova";
int n, i;
n = p - stringa (punto al 1°)

(punto dove c'è il carattere)

(cerco e stampo la stringa fino al carattere poi lo fa dopo)

for (i=0; i<n; i++) putchar (stringa[i]);
putchar ('\n'); printf ("%s\n", p+1)

(dopo lo stampo)

ALLOCAZIONE STATICA

fatta in modo rigido: il programmatore decide la dimensione e poi non si può cambiare

ALLOCAZIONE DINAMICA

fatta "run-time": x es se vedo da tastiera la dimensione del vettore

→ in questo caso uso solo esattamente la memoria che mi serve

→ x poi deve fare funzione malloc() che restituisce NULL se non è andata con successo, se no l'indirizzo di inizio
↳ (se x es non c'è abbastanza mem)

void * malloc (<dimensione>)
↓
1 solo parametro
(dimensione in byte)
↳ posso dichiarla con lo sizeof()