Appunti per il corso di

Algoritmi e Strutture Dati

1° anno - Laurea e Diploma in Informatica

Università di Genova

a.a. 1998/99

(autore: prof. Gerardo COSTA)
(docente: prof. Enrico PUPPO)

1ª Puntata

Programmazione 0.5

(versione: Marzo 1999)

Testi di riferimento (per il C)

[KR] Kerninghan - Ritchie: Il linguaggio C (versione ANSI) - editore Jackson

[DM] Darnell - Margolis: C Manuale di programmazione - editore McGraw-Hill

[KR] è il riferimento principale; [DM] è un'alternativa, forse più accessibile per chi non ha esperienze di programmazione,

Indice

1 . Premessa

Il titolo di questo corso è *Algoritmi e strutture dati*; la prima parte tuttavia andrebbe chiamata *Introduzione alla programmazione*. Cerchiamo di chiarire, almeno in parte, il significato di *algoritmo* e di *struttura dati* ed il collegamento con i *programmi*.

Il termine *algoritmo* deriva dal nome del matematico persiano Mohammed ibn-Musa *al-Khowarismi* vissuto nel IX secolo d.C. Lo usiamo per indicare una

"descrizione di una successione di *azioni* che permettono di risolvere un problema assegnato".

Un programma, scritto in un linguaggio di programmazione come il Pascal o il C, è un esempio di algoritmo in cui il livello di precisione e di dettaglio è molto alto; una ricetta di cucina è un altro esempio (legato a tutt'altro genere di problemi), con un livello inferiore di dettaglio e precisione.
Le *azioni* nei nostri algoritmi sono essenzialmente manipolazioni di *dati*, o *strutture dati*.

Un esempio dovrebbe aiutare a capire.
Supponiamo che ci venga assegnato il problema seguente:

dato un testo (ad esempio un articolo di giornale) stabilire quali sono le parole che vi compaiono con la frequenza massima (dove per parola si intende un sostantivo, o un aggettivo, o una preposizione, o un articolo,...)

Un modo di risolvere questo problema è:

- leggere il testo partendo dall'inizio e costruire, man mano, un *dizionario* (o elenco) delle parole che incontriamo; nel dizionario, ad ogni parola associamo un numero, o meglio un contatore, che registra quante volte abbiamo incontrato la parola, fino al quel momento;
- alla fine si tratterà solo di controllare nel dizionario, e stampare le parole il cui contatore ha valore massimo.

Dunque, a partire dai dati di ingresso, il testo, costruiamo una *struttura dati*, il dizionario; la soluzione del problema si ottiene trasformando (nel nostro caso: arricchendo) questa struttura dati e, alla fine, estraendone la risposta voluta.
Nell'esempio si vede come il procedimento per arrivare alla soluzione, cioè l'algoritmo, sia strettamente legato all'utilizzo di una particolare struttura dati, il dizionario; questo è un fatto del tutto generale.
Il problema considerato è semplice, tuttavia c'è almeno un punto che andrebbe precisato:

quand'è che due parole sono da considerarsi uguali ?

Detto in un altro modo: "cane" e "cani" sono due parole distinte o no? (analogamente per le coppie: "piove" e "pioveva", " po' " e "poco"; per non parlare di " l' " ....). Notiamo che questo non è un problema di informatica; la sua soluzione non va cercata sfogliando i libri di informatica, ma andando a parlare con chi ci ha assegnato il problema e chiedendogli cosa vuole veramente ("estrarre le informazioni dal cliente" è uno dei grossi problemi che affligge chiunque si occupi di sviluppo software su commissione). Facciamo l'ipotesi, che il cliente voglia tenere distinti "cane" e cani" ed anche "la", 'lo" e " l' ".
Il problema a questo punto è molto semplice e l'algoritmo per arrivare alla soluzione è altrettanto semplice, quindi la *correttezza* di quest'ultimo è immediata (purché non si facciano errori nella lettura e nella fase di individuazione delle parole e di confronto tra di esse); in genere non è così e la verifica della correttezza è un aspetto essenziale nel progetto di algoritmi.
Il nostra algoritmo è *ad alto livello*: utilizza l'idea di dizionario, senza precisare come questo è realizzato (quindi, in realtà stiamo lavorando con quello che viene detto *tipo di dato*); in una fase successiva bisognerà farlo, scoprendo che vi sono molti modi (molte strutture dati concrete).
Le *prestazioni* (si dice anche: l'*efficienza*, la *complessità*) del nostro algoritmo dipendono, essenzialmente, proprio da come viene organizzato il dizionario; in casi più complessi, l'efficienza dipende anche dal modo in cui si precisano i vari passi dell'algoritmo.
La fase finale è poi quella di tradurre in un linguaggio di programmazione effettivo il nostro algoritmo, ottenendo quindi un *programma*. (In realtà non è così perché nel frattempo il cliente ha cambiato idea ....)

Nel corso, per prima cosa vedremo i primi elementi di programmazione, partendo da zero, o meglio dando per note solo alcune nozioni di base viste nel corso di *Architettura degli elaboratori*. Poi

affronteremo gli altri aspetti: algoritmi e strutture dati per problemi “classici”, correttezza ed efficienza; non necessariamente in quest’ordine e senza nessuna pretesa di completezza.

Ci limitiamo a considerare la *programmazione imperativa, tradizionale* e *“in piccolo”*.

In piccolo (in the small, in cattivo inglese), si riferisce alla dimensione dei programmi, ma anche dei problemi che considereremo (per la precisione ci limiteremo a problemi e programmi piccoli-piccoli). Tradizionale, perché non consideriamo: parallelismo, concorrenza, non determinismo, oggetti. (Un esempio recente di linguaggio ad oggetti, con possibilità di programmazione concorrente, è Java; sul versante tradizionale abbiamo: FORTRAN, ALGOL, Pascal, C, ...)

Per chiarire il termine *imperativo*: i linguaggi del livello “macchina convenzionale” e del livello “assembler” , visti nel corsi di Architettura degli elaboratori, sono linguaggi imperativi. La caratteristica principale è che un programma è, essenzialmente, una successione di istruzioni; queste istruzioni hanno la forma di *comandi* (incrementa questo, shifta quest’altro, copia il contenuto di una cella di memoria in un’altra, salta al comando con etichetta K,.....). I linguaggi imperativi che si usano comunemente per programmare, ad esempio Pascal e C, usano istruzioni più complesse, più comode, che però mantengono la caratteristica di essere comandi. Risolvere un problema nello stile della programmazione imperativa significa trovare una successione di istruzioni tali che, se eseguite, producono la soluzione del problema. Quindi, un programma in stile imperativo descrive *un modo per arrivare* alla soluzione. Per contrasto, in altri stili di programmazione, in particolare la programmazione logica, un programma descrive *cosa si vuole*, lasciando all’implementazione del linguaggio stesso il compito di trovare un modo per arrivarci.

Queste questioni verranno approfondite nel corso di *Linguaggi di programmazione*, previsto al 2o anno (Laurea e Diploma), quindi ci fermiamo qui.

D’ora in avanti, per brevità: linguaggio = linguaggio imperativo (sequenziale, tradizionale).

Il linguaggio che usiamo per la parte pratica del corso (in particolare il laboratorio) è il C. Come tutti i linguaggi, ha le sue peculiarità. Non vogliamo enfatizzarle, in particolare vogliamo evitare che il C venga considerato “il modello di linguaggio di programmazione”. Quindi cercheremo di presentare i vari aspetti di un linguaggio (struttura dei programmi, dichiarazioni, istruzioni,...) in modo un po’ generale e valido almeno per i linguaggi affini al C, in particolare il Pascal (se non altro perché molti libri usano il Pascal come linguaggio per descrivere gli algoritmi e la realizzazione concreta delle strutture dati). Inoltre per descrivere gli algoritmi utilizzeremo un linguaggio a mezza strada tra: Pascal, C, linguaggio della matematica, linguaggio naturale; questo per due motivi: non legarci ad un particolare linguaggio, ma soprattutto poter utilizzare, a seconda dei casi, la notazione più comoda o più familiare.

Queste note fanno da supporto alla prima parte del corso: Introduzione alla programmazione. Qualche argomento è presentato in modo abbastanza dettagliato, tutti gli altri sono trattati in modo sbrigativo, anche molto sbrigativo (come ad esempio le espressioni), con molte semplificazioni.

Quindi, in generale, queste note NON costituiscono un riferimento per i vari concetti legati alla programmazione.

Inoltre è previsto che le lezioni cui si riferiscono queste note siano affiancate da altre dedicate in modo specifico al C (per queste il riferimento è [KR], o [DM], e materiale disponibile “in linea”); quindi da un certo punto in poi, diamo per scontato che alcuni aspetti del C siano noti e facciamo riferimento ad essi senza spiegarli.

2 . Struttura dei programmi

Ricordiamo che qui ci limitiamo ai linguaggi *imperativi* e *tradizionali,* con particolare riferimento al C ed al Pascal.

Un *programma* è composto da uno o più *moduli*, o *unità* (in C: unità = file).

Per ora ci limitiamo a programmi formati da una sola unità.

Una *unità* è composta da intestazione(nome + info per collegarsi ad altre unità + ...)
parte dichiarazioni
parte istruzioni

Un esempio: programma per trovare i numeri primi da 0 a 100.

Prima di tutto:

- n è primo se gli unici divisori sono 1 ed n;
- k divide n sse n mod k = 0 (cioè, il resto della divisione di n per k è 0);
- sappiamo che 1 e 2 sono primi;
- se n ≥ 3, per capire se è primo, basta controllare se è divisibile per k = 2, 3, ..., r
  dove r è la parte intera superiore della radice quadrata di n; quindi ci si ferma appena si trova un k che divide n, oppure tale che k^2 supera n.

Riportiamo, senza spiegarli, due programmi, uno in Pascal, l'altro in C, basati su quest'idea; andando avanti il significato delle singole componenti dovrebbe chiarirsi.

Si vede subito che le tre parti sono ben distinte nel programma in Pascal, mentre non è così in C; ritorneremo su questo più avanti, dopo aver parlato di visibilità delle dichiarazioni.

Programma in Pascal (commenti ***in corsivo***)

```
program  primi(input, output);  questa è l'intestazione

parte dichiarazioni:

 const  MAXNUM = 100;
 var  n,k : integer;
     ancora : boolean;

parte istruzioni:

 begin
  writeln (1, "  e` primo");
  writeln (2, "  e` primo");

  for n := 3 to MAXNUM do
       begin
          k := 2;    ancora := true;
          while  (k*k <= n)  and  ancora  do
               begin
                  if  (n mod k) = 0  then  ancora := false;
                  k := k+1
               end;
          if  ancora  then writeln (n, " e` primo")
       end

 end.
```

Programma in C

```
#include <stdio.h>
#define  MAXNUM  100

main()
{ int n,k;

  printf("%d e` primo\n",1);
  printf("%d e` primo\n",2);

  for(n=3; n<=MAXNUM; n++)
   {
        for(k=2 ; k*k<=n ; k++)   if(n % k == 0) break;

        if  (k*k > n)   printf("%d e` primo\n",n);
    }
}
```

3 . Due parole sui “tipi”

Nei linguaggi di programmazione moderni il concetto di “tipo di dato” è uno dei concetti chiave. Cercheremo di chiarirlo nella parte di dispense dedicate, appunto, ai tipi di dato. Tra l'altro, vedremo la differenza tra “tipo” e “tipo di dato”; per ora, tuttavia, possiamo confondere le due cose (come del resto si fa spesso nei libri che parlano di linguaggi di programmazione).

Qui qualche cosa; male che vada, per l’uso che ne faremo al momento, il significato comune della parola “tipo” è sufficiente.

Esistono tanti tipi di lampadine; analogamente in Pascal, e in misura minore il C, offrono una scelta di tipi (di dato).

In primo luogo, quelli di base, predefiniti, cioè già pronti per l’uso; ad esempio, gli interi (il tipo `integer`) ed i razionali (il tipo `real` del Pascal ed il tipo `float` del C). Questo significa che in un programma è possibile utilizzare i numeri interi e razionali (meglio: un sottinsieme di tali numeri; come visto nel corso di Architettura degli elaboratori) e le principali operazioni su di essi. Più precisamente, il linguaggio offre (almeno):

- delle costanti come: 17 e -15 (che sono intere), 3.14 e -0.5 (che sono razionali);
- delle operazioni sugli interi e sui razionali: somma, moltiplicazione, uguaglianza,...

In secondo luogo, come vedremo, il programmatore può definire, o costruire, tipi più complessi; come minimo tipi che corrispondono all’idea di successione, di vettore, di tabella, di scheda anagrafica,....

4 . Dichiarazioni

In un modo o in un altro nei vari linguaggi si hanno *dichiarazioni* di:

- costante (vedi sotto)
- variabile (vedi sotto)
- tipo (vedremo più avanti)
- procedura / funzione (vedremo più avanti)

Scopo delle dichiarazioni è: introdurre delle abbreviazioni, o dei sinonimi (es.: le dichiarazioni di costante); introdurre delle definizioni (es. le dichiarazioni di funzione, analoghe a quelle che si usano in matematica: sia f la funzione .....); dichiarare (nel senso comune delle parola) che nel programma ci serviranno certi oggetti di un certo tipo (es. le dichiarazioni di variabile). Questo serve all’implementazione per poter predisporre le cose nella maniera migliore e a chi legge per orizzontarsi.

4.1. Dichiarazioni di costante

Sostanza:

si dichiara (nel senso comune della parola) che un nome (o più precisamente: *identificatore* - vedere. oltre) corrisponde ad una costante; esempio: " useremo pg per indicare 3.14159 "
quindi: usare pg oppure 3.14159 è lo stesso, ma scrivere pg è più comodo.

Forma:

Pascal	`const pg = 3.14159;`
C	`#define pg 3.14159`
noi	come Pascal, o a parole

Nota. Per quello che riguarda il C (ed il `#define`) le cose sono un po' più complicate, ma al momento non ci interessa approfondire le questione.

4.2. Dichiarazioni di variabile

Più complesse da spiegare di quelle di costante; per il momento accontentiamoci di dare un'idea delle dichiarazioni di *variabili semplici*.

Forma:

Pascal	`var x : integer;`
C	`int x;`
noi	come Pascal, o a parole

Sostanza:

l'effetto (in Pascal, C, ...) della dichiarazione di sopra è il seguente:

- si riserva in memoria una *cella-per-un-intero* (= spazio di memoria di dimensione fissata, e dipendente dal linguaggio, sufficiente a contenere un valore di tipo intero); chiamiamola Cx ; invece di cella, si usa anche la parola *locazione*;
  chiaramente se avessimo dichiarato: var x: real, la cella sarebbe stata di dimensione sufficiente per un razionale, e così per gli altri tipi;
- si ricorda l'associazione: x ↔ Cx

d'ora in avanti (a meno da non ri-dichiarare x - vedere oltre: Visibilità delle dichiarazioni) il valore di x (meglio: del nome x) è, in base al contesto:

l'indirizzo di Cx (detto anche *valore sinistro (left value) di x*, o *l-value di x*) oppure

il contenuto di Cx (detto anche *valore destro (right value) di x*, o *r-value di x*)

Punto chiave: l'indirizzo non cambia, il contenuto può cambiare (vedere anche istruzione di assegnazione).

Note.

- La ***variabile*** è la cella Cx , mentre x è il *nome di variabile* (ed è un *identificatore* - vedere sezione seguente). Spesso, si confondono le due cose e si dice "la variabile x"; non è grave, ma bisogna aver chiara in testa la differenza.
- La cella non è un indirizzo, un byte, una "parola", o altro, nella macchina fisica, ma è un qualcosa a livello di "macchina virtuale C, o Pascal, o .....". In questa macchina immaginiamo di avere celle per interi, celle per caratteri, ..., celle per i vari tipi che definiamo, con la dimensione giusta (e quindi anche un qualche controllo sui valori che possono contenere) ed un indirizzo.

Un esempio semplice per capire il ruolo delle variabili.

Problema: dato un intero n, n≥1, voglio trovare k tale che $2^k \le n < 2^{k+1}$;
quindi k = parte intera di $\log_2 n$.

Algoritmo: *parte dichiarazioni:*

dichiariamo due variabili kk , p intere

(alla fine kk ci fornirà il valore k voluto)

parte istruzioni:

partiamo con kk = 0 e p = 1 *(quindi: $p = 2^{kk}$)*

[#] se p ≥ n allora se p = n allora scrivi kk *ed esci*
altrimenti scrivi kk-1 *ed esci*

altrimenti incrementa (di 1) kk
moltiplica p per 2 *(quindi: $p = 2^{kk}$)*
e riprendi da [#]

Come si traduce l'algoritmo di sopra in un programma vero e proprio ? Lo vedremo presto; ora ci interessa osservare che la parte istruzioni contiene delle *espressioni* (p=n, kk-1,...).

Prima di parlare di istruzioni dobbiamo quindi dire qualcosa sulle espressioni.

5 . Espressioni

In ogni linguaggio ci sono:

- valori / costanti; ad esempio:
  -17 valore intero
  3.14 valore "reale" o meglio: razionale
  'a' valore carattere
  true valore booleano (di verità)
- identificatori (nomi semplici), quasi sempre della forma:
  una lettera (maiuscola o minuscola), oppure
  una lettera (maiuscola o minuscola) seguita da lettere / cifre / altri caratteri (non tutti questi sono legali, dipende dal linguaggio; il C ammette solo ' _ ')
  esempi: kk p size_of x77 Max EOF NULL
- simboli di operazione
  aritmetici: + * / % .....
  logici: and or not .....
  relazionali: = < ≤ .....

Tra i vari linguaggi, ci sono delle *differenze sintattiche*, cioè differenze nel modo di scrivere le cose (ad esempio: il simbolo di uguaglianza si scrive = in Pascal, ma == in C; i razionali si chiamano `real` in Pascal e `float` o `double` in C;.....), ma anche delle differenze più significative, come vedremo andando avanti. Per fortuna, per quello che ci serve, si rimane sempre abbastanza vicini allo stile matematico usuale.

Usando le costanti, gli identificatori e i simboli di operazione, si formano (più o meno nel modo solito) le **espressioni**.

Esempi: 17 + 3 x - 5 x3 - 5 A99 / 77 +4
(aa + bb) / cc ((x + y) = z) or ((x > 2) * y)

Sono anche previste certe convenzioni standard (*regole di precedenza*); ad esempio:

x + y * z equivale a x + (y * z) dove * indica la moltiplicazione.

Diamo solo alcune indicazioni per le classi di espressioni che useremo più frequentemente, riferendoci al C ed al Pascal; per tutto il resto: **imparare a consultare i manuali !**

Espressioni aritmetiche intere

Useremo la notazione decimale solita.

Come operazioni, abbiamo a disposizione almeno: +, * (indica la moltiplicazione), - , divisione intera (`div` in Pascal, / in C; ad esempio: 10 / 3 è 3), modulo, o resto della divisione intera (`mod` in Pascal, % in C; ad esempio: 10 % 3 è 1).

Per le potenze ci si deve arrangiare: a^3 diventa a*a*a mentre a^{27} si ottiene tramite un "pezzo di programma", o meglio utilizzando una *funzione* (e saremo presto in grado di capire come).

Bisogna ricordare che c'è un limite alla dimensione degli interi: in base al linguaggio, alla macchina, ..., per gli interi si usano 16, 32, 64, ... bit, quindi c'è il pericolo di *overflow* (vedere appunti di Architettura degli elaboratori). Un'implementazione attuale del C permette comunque di usare almeno interi da $-2*10^6$ a $2*10^6$.

Espressioni aritmetiche razionali

Useremo la notazione decimale solita, col il punto al posto della virgola.

Come operazioni, abbiamo almeno: +, * , - , / (/ indica la divisione).

Per le potenze, come sopra; per radici quadrate, funzione trigonometriche, logaritmi, ... sia in Pascal che in C ci sono delle *funzioni di libreria*, che si possono utilizzare nei programmi, senza dover faticare troppo.

I limiti alle dimensioni dei numeri ci sono anche qui (vedere appunti di Architettura degli elaboratori) ma in C , come minimo, si va da -10^{37} a 10^{37}; quindi il problema vero è quello della *precisione*: (10.0 / 3.0) * 3.0 è equivalente a 10.0 come dovrebbe ? Purtroppo non c'è garanzia! In particolare, il test (10.0 / 3.0) * 3.0 = 10.0 tipicamente produce risultato falso.

La questione è abbastanza complicata, perché nel valutare una espressione complessa gli errori prodotti dalle singole operazioni si accumulano e si amplificano; qualche aspetto verrà discusso nel corso di Calcolo numerico. Qui possiamo solo segnalare che il problema esiste; non aspettiamoci dunque risultati 'esatti" quando calcoliamo espressioni razionali.

Interi e razionali normali, corti, lunghi, lunghissimi, con segno, senza segno.

Per permettere, a seconda delle necessità, un intervallo di valori più piccolo/grande o una minore/maggiore precisione, il C prevede diverse varianti di interi e di razionali; sono aspetti che qui trascuriamo.

Come in matematica, si possono mescolare interi e razionali: se scriviamo 10.2 + 3 otteniamo 13.2. Sui manuali del C ci sono pagine dedicate a spiegare nei dettagli cosa succede quando si moltiplica un intero corto con un razionale lungo .... Qui adottiamo lo "stile matematico" senza preoccuparci dei dettagli.

Espressioni booleane

In Pascal hanno identità e dignità pari a quella delle espressioni aritmetiche; in C invece si mascherano da intere.

In Pascal, ci sono:

- identificatori booleani; valori booleani: `true` `false`
- tre connettivi: `and` `or` `not`
- infine, se exp_1 , exp_2 sono due espressioni intere, o razionali, ... e `rel` è uno tra = , < , > , <> , <= , >= (dove gli ultimi tre corrispondono a: ≠ , ≤ , ≥), allora exp_1 `rel` exp_2 è una espressione booleana.

In C tutto questo si "maschera", si simula, usando gli interi; vediamolo con una tabellina.

Pascal	C	note
true	1	anche qualunque valore diverso da 0
false	0	
var bb boolean	int bb	ma anche char bb
and or not	&& || !	
= <>	== !=	
< <= > >=	< <= > >=	

Se, come faremo noi, si ragiona "come in matematica" e poi si traduce in C o Pascal, i due linguaggi sono perfettamente equivalenti ed equivalenti "alla matematica", salvo che per un punto: i connettivi and e or del C sono valutati rigidamente da sinistra a destra come illustrato dalle tabelle seguenti:

	exp_1	exp_2	exp_1 && exp_2	note
valori:	falso	???	falso	exp_2 **non** viene valutata
	vero	x	x	exp_2 viene valutata

	exp_1	exp_2	exp_1 || exp_2	note
valori:	falso	x	x	exp_2 viene valutata
	vero	???	vero	exp_2 **non** viene valutata

La differenza con l'and e l'or della logica, si ha nei casi in cui exp_2 non viene valutata: è possibile avere un risultato (vero o falso) anche nel caso che exp_2 sia "priva di senso".

Nel seguito scriveremo le espressioni usando una notazione matematica abbastanza standard (quindi usiamo = per l'uguaglianza, ≠ per diverso, ...), senza preoccuparci di come si traduce in C o Pascal (anche perché in genere la traduzione è banale); però useremomo * per indicare la moltiplicazione.

6 . Istruzioni

6.1. Assegnazione

E` l'istruzione tipica per modificare il valore di una variabile, nel senso di cambiarne il contenuto.

La **forma** più **semplice** è schematicamente (vedi spiegazione sotto):

Pascal < identificatore> **:=** <espressione>

C < identificatore> **=** <espressione> **;**

noi < identificatore> ← <espressione>

Esempi:

Pascal	x := 2+2	area **:=** base * altezza / 2
C	x = 2+2;	area **=** base * altezza / 2 **;**
noi	x ← 2+2	area ← base * altezza / 2

L'identificatore deve essere un **identificatore di variabile**, precedentemente dichiarato come tale.

Significato. Supponiamo: x e y variabili intere, z variabile booleana. Allora:

Assegnazione	**Effetto**
x ← 3	mettiamo 3 nella cella che corrisponde ad x
x ← 3*2	valutiamo 3*2 e mettiamo il risultato nella cella per x
x ← y + 1	valutiamo y, cioè andiamo a leggere il contenuto della sua cella, a questo sommiamo 1 e mettiamo il risultato nella cella per x; sarebbe più chiaro, ma più verboso, se si scrivesse qualcosa come: x ← contenuto(y) + 1
x ← x - 3	del tutto analogo al precedente (quando si scrive come in C: x = x - 3 fa più impressione)
z ← x > 0	infatti, x > 0 è una espressione booleana

Nell'esempio x ← x-3 si vede chiaramente la distinzione tra valore sinistro e valore destro: per la x che è a sinistra della freccia interessa il valore sinistro, cioè l'indirizzo della locazione (cella), per la x che è a destra interessa il valore destro, cioè il contenuto di quella locazione.

Nota. In C, a dire il vero, l'assegnazione è prima di tutto una espressione (il cui valore è quello della espressione a destra e che ha come "effetto collaterale" quello ci modificare il valore della variabile a sinistra); diventa una istruzione aggiungendo il punto e virgola in fondo. Qui ignoreremo questo aspetto, usando l'assegnazione solo come istruzione.

Nota sul modo di rappresentare schematicamente istruzioni (ed altro)

La forma delle istruzioni (delle espressioni, delle dichiarazioni) varia, ma c'è sempre:

- una parte "fissa", che identifica il tipo di istruzione; qui: i simboli **=** e **;** per il C, il simbolo (composto) **:=** per il Pascal, il simbolo ← che useremo noi;
- una parte "variabile", qui l'identificatore e l'espressione.

Almeno all'inizio, nella descrizione delle istruzioni, useremo la convenzione di scrivere in grassetto le parti fisse e tra parentesi ad angolo il nome o una descrizione delle parti che variano.

Inoltre daremo: la forma tipica in C e Pascal e quella che useremo noi per presentare gli algoritmi.

Per un altro esempio, ritorniamo alle dichiarazioni di variabile; la forma generale in Pascal è:

var <identificatore> **:** <tipo>

Non si guarda avanti

In matematica non c'è niente di male a scrivere

a = 2b dove

b = 3.14

In altre parole, si può usare un simbolo e poi precisarne il significato o il valore.

Non vale l'analogo in C o Pascal (e in altri linguaggi):

a ← 2*b

b ← 3.14 NON va bene.

Come regola generale (che verrà precisata in seguito - vedere Visibilità delle dichiarazioni), in ogni punto del programma si può usare solo quello che è stato definito precedentemente.

6.2. Istruzioni di I/O (Input / Output)

Vediamo direttamente un paio di esempi, il primo decisamente stupido.

Problema 1: dati due numeri interi a e b; vogliamo metterli in ordine crescente:
a, b se $a \le b$
b, a se $a > b$

Vogliamo un unico programma che funzioni per ogni scelta di a e b. Quindi, lo schema è:

input: a b ⟶ PROGRAMMA ⟶ *output:* a b *oppure:* b a

L'input "sta fuori", su file o dato da tastiera; l'output, prima o poi "deve uscir fuori", su file o sul video.

Algoritmo:

dichiarazioni: due variabili intere x, y

istruzioni: *commenti*

leggi(<sorgente>, x) *leggi il 1o numero (a) da <sorgente>, file o tastiera, e mettilo in x (cioè nella cella associata ad x); notare che, nell'istruzione, a non viene citato;*

leggi(<sorgente>, y) *leggi il 2o numero (b) da <sorgente> e mettilo in y; nemmeno b viene citato (vedere nota).*

se $x \le y$
allora scrivi (<destinazione>, x, y)
altrimenti scrivi (<destinazione>, y, x)

Note.

1. Le istruzioni di input sono "leggi(.....)", quelle di output sono "scrivi(....)".

2. Le istruzioni usate (qui e in seguito), non appartengono a nessun linguaggio di programmazione, ma seguono lo schema di istruzioni effettivamente presenti in linguaggi come il Pascal o il C, quindi sono facilmente e direttamente traducibili in istruzioni autentiche.

Ad esempio, l'ultima istruzione in Pascal si scriverebbe:

```
if x <= y then write(nfd, x, y) else write(nfd, y, x)
```

avendo scelto nfd come nome del file destinazione.

3. Notare che `if x <= y then write(nfd, x, y) else write(nfd, y, x)`

è una singola istruzione, anche se contiene altre istruzioni al suo interno; lo stesso vale per

se $x \le y$ allora scrivi (<destinazione>, x, y) altrimenti scrivi (<destinazione>, y, x);

la situazione è del tutto analoga a quella delle espressioni: a + b/c è una singola espressione, anche se contiene delle altre espressioni, come b/c

4. Nelle istruzioni di lettura non c'è nessun riferimento alla "cosa da leggere" (qui: a, b); l'idea è che le cose da leggere stanno in fila da qualche parte (in un file), o verranno prodotte in successione (da tastiera); la prima istruzione di lettura preleva la prima "cosa" (qui: a), la seconda istruzione preleva la seconda cosa (qui: b) e cosi' via. Ma come si fa a sapere come sono fatte le cose da leggere ? Vediamo ad esempio l'istruzione leggi(<sorgente>, y). Bisogna leggere la prima "cosa" di <sorgente> e metterla nella cella per y; poiché y è intera, la "cosa" deve essere un intero.

Notare che l'istruzione leggi(<sorgente>, y) comporta un'assegnazione di un valore ad y, quindi y deve essere una variabile; non è così per le istruzioni di scrittura, come vedremo.

Suggerimento: rivedere la parte che riguarda l'input/output nella macchina VM2 di Architetture.

Problema 2 Vogliamo calcolare la media aritmetica di n numeri razionali. Dunque:

in input abbiamo: n, a_1, ..., a_n (supponiamo n>0)

in output vogliamo: (a_1 + ... + a_n) / n

Problema (nel senso: c'è un problema nello scrivere l'algoritmo per il Problema 2): non possiamo usare n variabili, perché non conosciamo il valore di n quando scriviamo l'algoritmo. Ritorneremo su questo parlando di array e liste; qui il problema in realtà non esiste: basta leggere gli a_i uno alla volta e calcolare, man mano, la somma parziale.

Algoritmo:

dichiarazioni: una variabile intera nn (per tenere il valore di n)

due variabili razionali: a (per i valori successivi degli a_i) e somma

istruzioni:		*commenti*
leggi(<sorgente>, nn)		
somma ← 0.0		*diamo un valore iniziale a somma*
ripeti nn volte	{ leggi(<sorgente>, a) somma ← somma + a }	
		a questo punto: somma = (a_1 + ... + a_n)
scrivi (<destinazione>, somma / nn)		*vedi Nota 3.*

Note.

1. La prima volta che si esegue leggi(<sorgente>, a) l'effetto è di assegnare ad a il valore a_1, quando si esegue la 2a volta si assegna ad a il valore a_2 e così via.

2. Le parentesi { } qui servono a delimitare la parte di programma che va ripetuta nn volte. Più in generale servono, come nelle espressioni, ad incapsulare una successione di istruzioni in modo da trattarle come una singola istruzione, ved. anche oltre: Blocchi.

3. L'istruzione scrivi (<destinazione>, somma / nn) equivale a:
valuta l'espressione somma / nn e scrivi il valore ottenuto su <destinazione>.

Forma delle istruzioni di I/O:

- noi useremo una forma semplificata:
  leggi (var_1, ..., var_k) dove var_1, ... sono variabili
  scrivi (exp_1, ..., exp_j) dove exp_1, ... sono espressioni
  trascurando, in genere, di specificare <sorgente> e <destinazione>
- per poter leggere programmini Pascal, basta sapere che le istruzioni equivalenti sono:
  read ([qui qualche volta si specifica un file], var_1, ..., var_k) oppure
  readln (.....) *dopo aver letto k dati si va a capo*
  write ([qui qualche volta si specifica un file], exp_1, ..., exp_m) oppure
  writeln (.....) *dopo aver scritto m valori si va a capo*
- per il C rimandiamo alle lezioni sul C e a [KR], oppure [DM].

6.3. Blocchi

Per semplificarci la vita nel seguito, definiamo:

<blocco> = una singola istruzione, oppure
una successione di istruzioni racchiusa tra parentesi { }

Vedremo poi che, almeno in C, i blocchi possono contenere anche dichiarazioni.

In Pascal si usa **begin** invece di { ed **end** invece di }.

A proposito di successioni di istruzioni: in genere le scriveremo "in colonna", esempio:

x ← 3

y ← 5

z ← x + y

però alle volte, per risparmiare spazio, le scriveremo "fianco a fianco" separate da un punto e virgola (e da spazi), esempio:

x ← 3 **;** y ← 5 **;** z ← x + y

Nota. Nei linguaggi di programmazione, a partire dall'ALGOL '60, il punto e virgola ha assunto un'importanza veramente "esagerata". Con l'esperienza (e consultando i manuali e facendosi aiutare da un buon compilatore) si impara a mettere i punti e virgola là dove servono e ad evitarli dove creano guai; nello scrivere gli algoritmi, invece, ce ne preoccuperemo pochissimo.

C'è poi la questione degli 'spazi bianchi"; nei linguaggi di programmazione ad alto livello, grosso modo, la regola è: fate come vi pare (mettetene quanti volete, anche nessuno), ma:

- niente spazi all'interno dei "simboli composti",
  sia predefiniti, ad esempio **:= == begin var while**
  che definiti dal programmatore, tipicamente i nomi, come somma;
  quindi NON va bene scrivere: **: =** **beg in** som ma
- almeno uno spazio per separare due identificatori, ad esempio, tra var e l'identificatore di variabile che segue; quindi
  var somma va bene
  var somma va bene
  varsomma NON va bene

6.4. Istruzioni di scelta (o condizionali)

Le abbiamo già usate: se .... allora ..... altrimenti.

Forma 1

Pascal: **if** <condizione> **then** <blocco 1> **else** <blocco 2>

C: **if (** <condizione> **)** <blocco 1> **else** <blocco 2>

noi: come Pascal, oppure: **se** <condizione> **allora** <blocco 1> **altrimenti** <blocco 2>

dove <condizione> è: una espressione booleana, cioè a valore vero/falso, in Pascal e per noi;
una espr. a valore intero (0 ↔ falso, ogni altro valore ↔ vero) in C

Forma 2

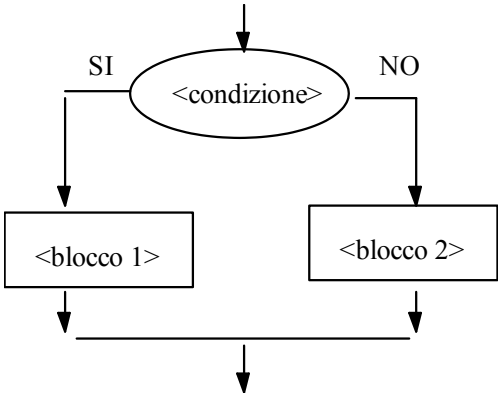
Come Forma 1, ma senza la parte **else** <blocco 2>

Significato (vedere anche disegno)

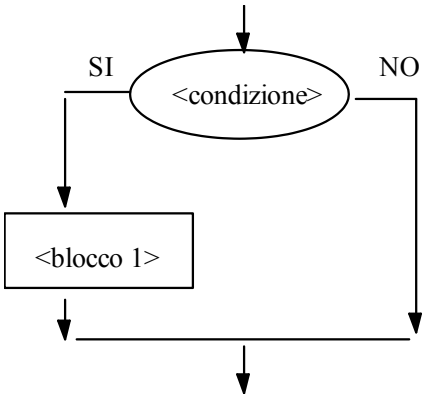
Forma 1. Si valuta la condizione; se il valore è "vero" allora si esegue <blocco 1> e si ignora <blocco2>; altrimenti si esegue <blocco 2> e si ignora <blocco 1>.

Forma 2. Si valuta la condizione; se il valore è "vero" allora si esegue <blocco 1>, altrimenti si passa a quello che segue.

Schema dell'istruzione **if-then-else** (SI = vero, NO = falso):



Schema dell'istruzione **if-then**



Esempi.

Abbiamo già visto a cosa serve la forma 1, vediamo un esempio per la forma 2.

Problema in input abbiamo: $n, a_1, ..., a_n$ (supponiamo $n>0$ e gli a_i interi)

in output vogliamo $\max \{ a_1, ..., a_n \}$ cioè il valore massimo degli a_i

Algoritmo.

dichiarazioni: 3 variabili intere: nn (per il valore di n), a (per i valori successivi degli a_i), max (per tenere il "massimo corrente", cioè il massimo dei valori già letti)

istruzioni: *commenti*

(1) leggi(nn) *ora nn vale n*

(2) leggi(a) *ora a vale a_1*

(3) max ← a *il massimo dei valori letti fino ad ora è senza dubbio a*

(4) ripeti nn-1 volte

{ (4.1) leggi(a)

(4.2) se a > max allora max ← a *altrimenti va bene il vecchio valore di max*

}

(5) scrivi (max)

Note

1. I numeri tra parentesi servono solo ad identificare le istruzioni.
2. Punto chiave per capire il programma e convincersi che funziona (esempio di un semplice *ragionamento di correttezza*):

dopo aver eseguito l'istruzione (3) il valore di max è effettivamente uguale al massimo dei valori letti fino al momento (uno solo); ogni volta che si esegue l'istruzione (4.2), meglio: dopo averla eseguita, ancora il valore di max è uguale al massimo dei valori letti fino al momento. Allora, quando arriviamo al punto (5), max è effettivamente il massimo di tutti i valori in input.

6.5. Istruzioni iterative

Abbiamo visto che serve poter scrivere: ripeti n volte { ..... }.

In molte situazioni, ci vuole qualcosa di più sofisticato. Il C ed il Pascal (ed altri linguaggi) offrono tre istruzioni: while, repeat-until, for (il Pascal); while, do-while, for (il C).

Il while del C e quello del Pascal sono identici nella sostanza, cambia solo leggermente il modo di scriverli; repeat-until e do-while sono anch'essi identici nella sostanza, ma con una differenza nella forma che è fastidiosa; le due istruzioni for sono invece diverse: quella del C è molto più flessibile e "potente".

In effetti, basterebbe solo il while, perché con esso si possono simulare le altre istruzioni; queste vengono offerte solo per comodità di programmazione e per favorire la leggibilità dei programmi.

6.5.1 . While

Forma del while

Pascal: **while** <condizione> **do** <blocco>

C **while (<condizione>)** <blocco>

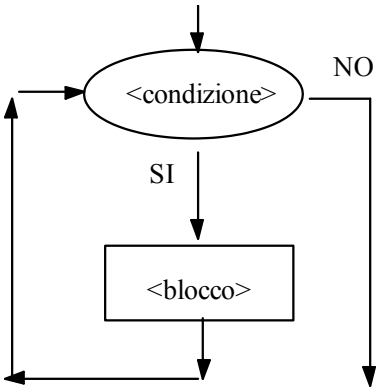
noi come Pascal.

con <condizione> come per if-then-else; il <blocco> è detto anche *corpo* (*body*) del while.

Significato (vedere anche disegno):

si valuta la condizione; se il valore è **falso** si continua, ignorando il corpo; se è **vero** si esegue il corpo e si ricomincia da capo (rivalutando la condizione ....); notare che se si vuole "uscire dal while", prima o poi bisogna che il corpo modifichi il valore della condizione, rendendola falsa; notare anche che la condizione potrebbe essere falsa subito.

Schema dell'istruzione while:



Vediamo per prima cosa come si ottengono le "ripetizioni" più semplici.

Per ottenere ripeti n volte <blocco> serve una variabile ausiliaria, che funge da contatore, sia cont:

```
        cont ← 1
        while
cont ≤ n  do  {
    <blocco>

                cont
←  cont + 1 }
```

Per un esempio un po' meno banale di uso del while, supponiamo di voler stampare una tabellina delle forma:

n	n^2	n^3	per n da 1 a MAX (costante)

Un semplice algoritmo (trascurando un po' i dettagli dell'output) è:

Algoritmo: *parte dichiarazioni:*

dichiariamo una variabile intera n

parte istruzioni:

ripeti per n = 1, 2, ..., MAX :

{ scrivi (n, n*n, n*n*n)

vai a capo sul file di output

}

Utilizzando un while, diventa:

Algoritmo: *parte dichiarazioni:*

dichiariamo una variabile intera n

parte istruzioni:

n ← 1

while n ≤ MAX **do**

{ scrivi (n, n*n, n*n*n)

vai a capo sul file output

n ← v + 1

}

Come esempio più interessante rivediamo il problema di calcolare la parte intera del logaritmo in base 2 di n.

Problema: abbiamo in input un intero n, n≥1

vogliamo in output l'intero k tale che $2^k \le n < 2^{k+1}$; cioè k = parte intera di $\log_2 n$.

La soluzione che avevamo visto (ignorando l'input/output) è:

Algoritmo: *parte dichiarazioni:*

dichiariamo due variabili kk , p intere

alla fine kk ci fornirà il valore k voluto

parte istruzioni:

partiamo con kk = 0 e p = 1 *quindi: p = 2^{kk}*

[#] se p ≥ n allora se p = n allora scrivi kk *ed esci*
altrimenti scrivi kk-1 *ed esci*

altrimenti incrementa (di 1) kk
moltiplica p per 2 *quindi: p = 2^{kk}*
e riprendi da [#]

Un modo equivalente di formulare la parte istruzioni è:

partiamo con kk = 0 e p = 1

fintanto che n > p ripeti { incrementa (di 1) kk
moltiplica p per 2
}

quando arriviamo qui: n ≤ p

se p = n allora scrivi kk *ed esci*
altrimenti *(cioè: n<p)* scrivi kk-1 *ed esci*

A questo punto è facile formulare il tutto usando un while (e if-then-else), aggiungendo anche un controllo sull'input:

Algoritmo:

dichiarazioni: 3 variabili nn, kk , p intere

istruzioni:

leggi(nn) *quindi nn ora ha per valore destro: n*

if nn < 1 then scrivi ("input errato") *fidarsi è bene, non fidarsi è meglio*

else { kk ← 0
p ← 1 *qui p = 2^{kk}*
while nn > p do { kk ← kk+1
p ← p*2 *qui p = 2^{kk}*
}
quando arriviamo qui: nn ≤ p
if p = nn then scrivi (kk)
else scrivi (kk-1)
}

Nota. Qui, ed anche in precedenza, abbiamo usato: n, k,... per il valori in input/output e nn, kk,... per le variabili di programma. Questo per sottolineare la differenza. Andando avanti tenderemo ad usare solo n, k,.....

Per esercizio: scrivere (nello stesso stile) un programma che legge n intero, n≥0, e scrive tutti gli interi, maggiori o uguali a 0, che sono potenze di 2 e sono < n.

6.5.2 . Repeat-until e do-while

Forma

Pascal: **repeat** <blocco> **until** <condizione>

C: **do** <blocco> **while (** <condizione> **)**

noi: non la useremo

con <condizione> come per if-then-else e while; il <blocco> è detto anche *corpo* (*body*).

Significato (vedere anche disegni)**:**

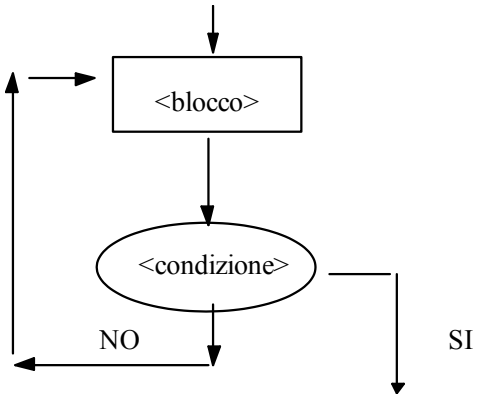
a differenza del while, prima si esegue il corpo e poi si valuta la condizione;

in Pascal: se il valore è **falso** si ricomincia da capo (eseguendo il corpo,....), se è **vero** si passa all'istruzione successiva; quindi l'idea è: "ripeti fino a quando la condizione non diventa vera";

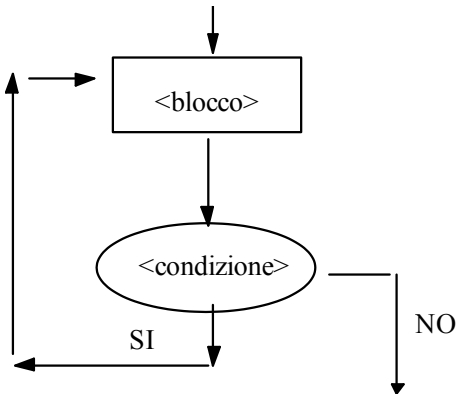
in C le cose funzionano al contrario: se il valore è **falso** si passa all'istruzione successiva, se è **vero** si ricomincia da capo (eseguendo il corpo,....); quindi l'idea è: "ripeti fintanto che la condizione rimane vera".

Proprio per questa differenza tra C e Pascal che potrebbe creare confusione, cercheremo di evitare di usare questo tipo di istruzione.

Schema dell'istruzione in Pascal:



Schema dell'istruzione in C:



Si vede subito che le due versioni si possono simulare usando un while; ad esempio (per il Pascal):

repeat <blocco> **until** <condizione>

è equivalente a:

{ <blocco>
while not <condizione> **do** <blocco> }

dove not è l'operazione di negazione logica.

6.5.3 . For

Abbiamo visto che per ottenere ripeti n volte <blocco> usando un while, ci vuole un po' di lavoro, ed una variabile ausiliaria; ma soprattutto si perde la semplicità e la chiarezza dell'originale.

Il Pascal ed il C, per questo motivo, offrono le istruzioni **for**.

La versione Pascal è più semplice, ma più limitata; quella del C decisamente più ricca, ma con la possibilità di scrivere cose incomprensibili. Qui diamo un'idea del for del Pascal; per il C rimandiamo alle lezioni sul C e a [KR] o [DM].

Il for del Pascal.

Assume due forme (cont è una variabili, detta *variabile di controllo*, tipicamente intera, o di tipo carattere, ..., exp1 ed esp2 sono due espressioni dello stesso tipo di cont):

- in salita: **for** cont **:=** exp1 **to** exp2 **do** <blocco>
- in discesa: **for** cont **:=** exp1 **downto** exp2 **do** <blocco

Il significato si spiega facilmente traducendo il tutto con una istruzione while; serve una variabile ausiliaria aux

- in salita: cont ← exp1
  aux ← exp2
  while cont ≤ aux do { <blocco> ; cont ← successore(cont) }
- in discesa: cont ← exp1
  aux ← exp2
  while cont ≥ aux do { <blocco> ; cont ← predecessore(cont) }

Il punto chiave è che le due espressioni che determinano i valori tra cui varia la *variabile di controllo* cont, vengono valutate all'inizio, quindi l'esecuzione del blocco non può influenzare tali valori; ad ogni passo il valore della variabile di controllo è incrementato, o decrementato, di "una unità" (+1, oppure -1 nel caso di interi; nel caso di caratteri, si passa al carattere successivo, o precedente, ...).

6.5.4 . Le nostre istruzioni

Per scrivere algoritmi vedremo di limitarci alle istruzioni seguenti:

- **while** <condizione> **do** <blocco>
- **per** <variabile di controllo> **=** <successione di valori> **:** <blocco>

Il while è quello visto.

L'istruzione **per ....** è non standard ed è stata scelta per poter esprimere in modo semplice quello che è semplice, senza doversi adattare alle regole del Pascal o del C. Il significato dovrebbe essere chiaro dagli esempi.; osserviamo comunque che nella <successione di valori> l'ordine è rilevante, come in tutte le successioni; quindi:

1, 2, 3 2, 1, 3 3, 2, 1 sono tre successioni diverse.

Inoltre, per l'istruzione **per ...**, supponiamo che il <blocco> non modifichi mai il valore della <variabile di controllo>, cioè si limiti ad usarlo.

Esempi:

1. **per** k **=** 1, 3, 5, ..., 117 **:** <blocco>

k assumerà, successivamente, i valori dispari da 1 a 117 (estremi inclusi) e il <blocco> verrà ripetuto per ciascuno di questi valori;

2. **per** k **=** n, n-1,...., 3 **:** <blocco>

k assumerà, successivamente, i valori da n a 3 (estremi inclusi), a scendere, e il <blocco> verrà ripetuto per ciascuno di questi valori; notare che se, all'inizio, n < 3, la successione è vuota ed il <blocco> non si esegue nemmeno una volta;

3. **per** k **=** 'a', 'e' 'i', 'o', 'u' **:** <blocco>

k assumerà, successivamente, i valori delle cinque vocali .....; notare che questa istruzione non si esprime facilmente con un while, o un for, se il blocco usa effettivamente il valore di k; il modo più semplice di tradurla è:

```
{   <blocco, con k='a'>
    <blocco, con k='e'>
    eccetera            }
```

Infine potrà capitare che la cosa più chiara sia un **for** stile C, allora lo useremo (naturalmente, dopo che questa istruzione è stata vista nelle lezioni sul C).

7 . Altre istruzioni.

Le citiamo semplicemente, con un minimo di spiegazione.

Goto Ha la forma goto <etichetta>. Va in coppia con la possibilità di mettere un'etichetta (numero o identificatore) davanti alle istruzioni, in modo da identificarle. L'effetto di goto <etichetta> è un salto incondizionato all'istruzione con l'etichetta <etichetta>.

I "salti all'indietro" servono per simulare dei cicli; avendo a disposizione istruzioni come for, while, repeat, questo modo di usare i goto è passato di moda. Restano i "salti in avanti', o i salti per terminare bruscamente un ciclo.

Per le uscite dai cicli è meglio usare istruzioni come "continue" o "break" (vedi sotto).

Per i salti in avanti: si può sempre rigirare la cosa usando un if-then-else; però in qualche caso questo può diventare pesante e allora ben venga un goto.

Break. Si usa in C nel corpo di un while, repeat o for, in genere controllato da una istruzione condizionale; ad esempio if .... then break .....

L'effetto di eseguire l'istruzione break è quello di uscire immediatamente dal ciclo (e quindi passare all'istruzione che segue il ciclo). Per un esempio, vedere il programma sui numeri primi nelle prima pagine di queste note; il confronto tra la versione Pascal e quella C dovrebbe chiarire l'uso del break.

Si usa anche, sempre in C, all'interno delle istruzioni di switch (vedi sotto).

Continue. Si usa come il break nei cicli, ma l'effetto è quello di interrompere l'esecuzione del ciclo corrente e passare al successivo. Non ci serve per quello che vedremo; per dettagli ed esempi si rimanda a [KR] o [DM].

Case (Pascal) e **switch** (C). Servono ad evitare lunghe cascate di if-then-else, allo scopo di rendere il programma più leggibile. Vedremo un esempio parlando di funzioni..

Return. Le vedremo parlando di procedure e funzioni.

Chiamate a procedura. Le vedremo parlando di procedure.

Istruzione nulla (o vuota). In qualche situazione, le regole del linguaggio richiedono la presenza di una istruzione, ma noi non "vogliamo fare nulla"; si usa allora l'istruzione nulla. In C questa corrisponde ad un ' ; ' isolato; ad esempio:

if (cond) **;** else <blocco> ; *che equivale a* if (!(cond)) <blocco> ;

In particolare, un ' ; ' di troppo spesso (ma non sempre !!!) non crea problema, in quanto viene interpretato come istruzione nulla.

8 . Array unidimensionali

Primo esempio di tipo di dato strutturato (o anche di *struttura dati*).

Corrispondono al concetto matematico di *vettore* (omogeneo).

Cominciamo con un esempio per capire il loro ruolo.

Vogliamo fare una statistica dei voti di maturità (da 36 a 60) degli iscritti al 1° anno di Informatica.

Con quanto visto finora, useremmo (tra l'altro):

- 25 variabili contatore intere: cont_36, cont_37, ..., cont_60
- 25 assegnazioni, per inizializzarle a 0
- eccetera ..... una noia mortale.

Per fortuna, si può dichiarare (proprio cosí in Pascal; in C si è costretti ad un piccolo contorcimento, come vedremo) una sola variabile cont come array di interi, con indici da 36 a 60:

var cont **: array [** 36 .. 60 **] of** integer

L'effetto di questa dichiarazione è:

- riservare una successione di 25 celle-per-intero: C_0, C_1, ..., C_24
- associare il nome cont all'intera successione (proprio così in Pascal; per il C vedremo meglio in seguito, per ora possiamo pensare che sia così anche in C);
- associare a ciascuna delle celle un nome della forma cont[k], precisamente:

  C_0 ↔ cont[36], C_1 ↔ cont[37], ... C_24 ↔ cont[60]
- d'ora in avanti, cont[36], cont[37], ..., cont[60] possono essere usati come qualsiasi altro nome di variabile intera: es. cont[38] ← 0 cont[40] ← cont[38] *2
- ma soprattutto è anche possibile usare "nomi generici", della forma:

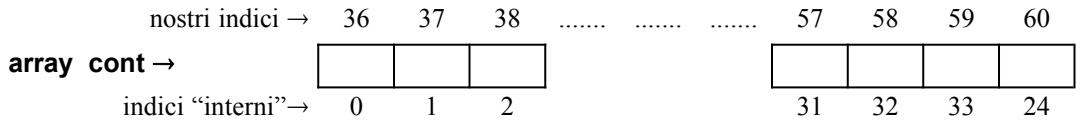
  cont [<exp>] dove <exp> è una espressione il cui valore deve essere un intero tra 36 e 60;

  cont [<exp>] equivale a cont [valore(<exp>)] ; ad es. se il valore di <exp> è 41, allora cont[<exp>] equivale a cont[41].

Quindi per inizializzare a 0 tutte le celle dell'array si può usare l'istruzione:

per k **=** 36, 37, ..., 60 **:** cont[k] ← 0.

Schematicamente, l'array cont si può rappresentare così:



Ritornando al nostro esempio:

Problema: in input abbiamo una successione di voti di maturità: v1, v2,....
in output vogliamo la tabellina: voto -- numero di persone con quel voto

Algoritmo: ignorando i dettagli di Input/Output, e supponendo di avere l'input nel file VOTI

variabili cont : **array [** 36 .. 60 **] of** integer
voto : integer

istruzioni

per k **=** 36, 37, ..., 60 **:** cont[k] ← 0

while (c'è ancora dell'input da leggere in VOTI)

do { (1) leggi(voto) *sottinteso: dal file VOTI*

(2) cont [voto] ← cont [voto] + 1

}

ora: cont[36] ci dice quanti sono i 36, cont[37] quanti sono i 37,.....

scrivi la tabellina

Nota. Uno dei problemi maggiori nell'usare i nomi come cont [<exp>] è che, se non si sta attenti, il valore di <exp> può non essere nell'intervallo voluto; nel nostro esempio, il file potrebbe contenere un errore, ad esempio un voto 25; quando nell'istruzione (1) leggiamo questo voto, la variabile voto assume valore 25; allora l'istruzione (2) equivale a

cont [25] ← cont [25] + 1

Questo è un pasticcio; l'effetto varia a seconda del linguaggio di programmazione usato (meglio: dell'ambiente di programmazione usato: Turbo Pascal, Turbo C, Code Warrior,....). Prima di tutto, l'errore non viene rilevato in fase di compilazione (a *compile time*, come si dice spesso), ma solo in fase di esecuzione (a *run time*). In fase di esecuzione, ci sono grosso modo 3 alternative:

un messaggio di errore e interruzione dell'esecuzione (caso migliore);

nessun messaggio ed un disastro incomprensibile dopo poco (caso medio);

tutto fila liscio, ottengo persino un output (caso peggiore! (perché poi?)).

Conclusione: bisogna che gestiamo noi la situazione, facendo dei controlli; ad esempio, nel programma di sopra, si potrebbe:

- aggiungere una variabile intera num_errori
- inizializzarla a 0 (prima del while)
- sostituire l'istruzione (2) con

if (voto ≥ 36) and (voto ≤ 60) **then** cont[voto] ← cont[voto]+1 **else** num_errori ← num_errori+1

Altro esempio, simile al precedente, di uso di array:

vogliamo leggere una frase e calcolare la frequenza delle lettere nella frase; per semplicità non distinguiamo tra maiuscole e minuscole.

Analogamente a prima, possiamo usare (almeno in Pascal) un array di interi, indiciato sulle lettere:

var frequenze **: array [** 'a' .. 'z' **] of** integer

Ultimo esempietto: prodotto scalare di vettori. Vediamo solo l'ossatura dell'algoritmo

costanti inf = 1, sup = 10

variabili aa, bb: **array [** inf .. sup **] of** real *questo è un esempio di dichiarazione multipla*

aa e bb sono 2 array ....

prod : real

j : integer

istruzioni: leggiamo i valori per aa e bb e li carichiamo in aa e bb *(omesso)*

prod ← 0.0

per j **=** inf, inf+1, ..., sup **:** prod ← prod + aa[j] * bb[j]

scrivi (prod)

Ricapitolando, in un array ad una dimensione abbiamo:

- *elementi*, tutti dello stesso "tipo" (intero o razionale negli esempi visti);
- *indici* (negli esempi: interi, lettere): valori contigui in un intervallo finito (per intenderci: questo intervallo deve poter essere messo in corrispondenza biunivoca con l'intervallo di interi 0 .. k , per qualche k); gli estremi dell'intervallo si chiamano anche *limiti dell'array*.

Una differenza tra C e Pascal sta negli indici:

in Pascal come sopra; in C: solo indici (interi) da 0 a k, per qualche k.

Inoltre, in C la dichiarazione è più concisa; ad es. la dichiarazione dell'array frequenze sarebbe:

```
int frequenze[26]
```

Questa dichiarazione specifica che frequenze è un array di 26 elementi interi (con indici da 0 a 25, estremi inclusi). Sta al programmatore stabilire la corrispondenza tra

```
lettere:      a  b  c       z
e indici: 0  1  2       25
```

Un'altra differenza riguarda la dimensione (cioè il numero di celle) degli array: il Pascal standard ammette solo array a dimensione fissata "al momento della compilazione", in C è possibile dimensionare gli array "al momento dell'esecuzione"; chiariremo questo punto negli esempi che seguono.

Per scrivere algoritmi useremo in genere gli array alla Pascal, ma senza esagerare nella scelta degli indici.

9 . Due esempi per "mettere tutto assieme"

9.1. Generare tutti i sottinsiemi di {1, ..., n}.

Un esempio, meno banale del solito, che mette assieme molto di quanto visto finora: generare tutti i sottinsiemi di A = {1, ..., n}.

Idea:

I sottinsiemi di A sono in corrispondenza biunivoca con i numeri binari di lunghezza n. Infatti:

- dato il numero s il sottinsieme corrispondente, chiamiamolo I(s) , si ottiene come segue:

  un "1" in posizione k (partendo da sinistra) indica che k è in I(s)

  uno "0" in posizione k (partendo da sinistra) indica che k non è in I(s) .
- rovesciando il discorso, dato B sottinsieme di A si trova il numero corrispondente, sia N(B);
- si vede facilmente che: N(I(s)) = s e I(N(B)) = B.

Per generare tutti i sottinsiemi di A, basta quindi generare tutti i numeri binari di n cifre.

Questo si può fare partendo da

00 .... 0 (n zeri) che corrisponde all'insieme vuoto

e continuando ad aggiungere 1 (con aritmetica binaria)

fino a 11 ... 1 (n "uni") che corrisponde a tutto A

In effetti è più comodo (per capire quando fermarsi) fare ancora un passo ed arrivare fino a

1 00 ... 0 (1 seguito da n zeri) *che non corrisponde ad un sottinsieme di A.*

Per capire come realizzare l'operazione "aggiungere 1" vediamo un es. con n = 6:

```
s      0 0 1 0 1 1 +
                   1
-------------------
r      0 0 1 1 0 0
```

r si ottiene da s partendo dal fondo, cambiando in 0 tutti gli 1 che si incontrano, fino a trovare il primo 0, che viene cambiato in 1.

Prima di vedere un possibile algoritmo, riformuliamo il problema in modo più sintetico:

Input: n (cioè la dimensione/cardinalità dell'insieme A)

Output: tutti i sottinsiemi di A (qui, però, non ci preoccupiamo di come vengono stampati)

L'algoritmo usa un array, che chiamiamo subset, per memorizzare i numeri binari di n+1 cifre.

Come detto sopra: all'inizio, in subset mettiamo il numero formato da n+1 cifre 0, poi man mano incrementiamo il numero di 1 ...

Poiché ci servono solo due valori, possiamo usare un array di booleani: true corrisponde a 1 e false corrisponde a 0.

Sarebbe bello procedere nel modo seguente:

prima leggere n e poi dichiarare subset come array [0 .. n] of boolean

Però non è possibile, almeno nei linguaggi che usiamo. Il Pascal standard non permette nulla del genere: gli estremi dell'intervallo degli indici devono essere valori costanti.

Nemmeno il C permette una dichiarazione equivalente a quella di sopra, cioè:

`int subset[n+1]` oppure `char subset[n+1]`

Per ottenere l'effetto voluto bisogna usare la corrispondenza tra array e puntatori; ma questo lo vedremo in seguito.

Quindi siamo costretti ad usare lo schema seguente:

- dichiariamo una costante, chiamiamola MAX, con un valore ragionevole;
- dichiariamo subset come array [0 .. MAX] of .... ;
- leggiamo n e lo confrontiamo con MAX; se è maggiore stampiamo un messaggio di errore e abbandoniamo, altrimenti facciamo quello che dobbiamo fare usando solo una parte dall'array: le prime n+1 posizioni;
- se poi scopriamo che il valore di MAX non va bene, modifichiamo il programma, cambiando solo la dichiarazione di MAX.

Fino a quando non avremo visto come aggirare l'ostacolo in C, seguiremo sempre uno schema di questo tipo.

Algoritmo

costanti: MAX =10 *è la dimensione massima degli insiemi* ;

10 non è poco: se A ha 10 elementi , i sottinsiemi sono 2^{10}, cioè più di 1000

variabili: n : integer *la dimensione effettiva dell'insieme A*

subset : array [0 .. MAX] of boolean

true = 1, false = 0

subset[0] corrisponde alla "posizione di overflow"; quando assume valore 1 siamo arrivati a generare 10...0

k : integer *per "scorrere" l'array*

istruzioni:

```
leggi (n)
if n > MAX  then  scrivi ( "errore, input troppo grande" )      e  si esce dal programma
else    {
            per  k  =  0, 1, ..., n :  subset[k] ← false   cioè: partiamo dal numero formato da tutti zeri
            while  not subset[0]                cioè fino a che non abbiamo generato 10...0
            do {
                    stampa ( l'insieme che corrisponde al numero contenuto in subset  da 1  a n)

                    quanto segue aggiunge 1 come visto prima:
                    k ← n
                    while  subset[k]  do  {  subset[k] ← false
                                             k ← k - 1
                        }
                    subset[k] ← true

             }      chiude il while più esterno
        }           chiude il ramo else
```

9.2. Selection Sort (ordinamento per "selezione").

Vediamo un algoritmo di ordinamento; qui ci serve per ricapitolare quanto visto, ma è interessante in sé; lo rivedremo meglio in seguito, parlando di *procedure*.

Problema: in input un intero $n \geq 1$ ed n numeri "reali" $a_1, ..., a_n$

in output $b_1, ..., b_n$ tali che la successione $b_1, ..., b_n$ è una permutazione della successione $a_1, ..., a_n$ ordinata in modo crescente ($b_1 \leq b_2 \leq ... \leq b_n$).

Idea dell'algoritmo:

1. si memorizzano i numeri da ordinare in un array, sia a, nelle posizioni da 1 a n
2. si riordinano gli elementi dell'array col metodo detto selection-sort:
   - 2.1. si cerca il valore minimo nell'array; supponiamo sia nel posto di indice k; si scambia a[1] con a[k]; ora il minimo è in prima posizione e quindi "a posto"; questo termina la "prima passata"
   - 2.2. si ripete quanto sopra considerando la parte di array da 2 a n
   - 2.3. e così via .... (vedere esempio sotto)
3. si stampano gli elementi di a (che ora sono in ordine).

Esempio con un array a di 7 elementi:

inizialmente, in a abbiamo:	5.1	1.5	0.7	9.0	0.9	7.3	1.5
dopo la 1ª passata, in a abbiamo:	0.7	1.5	5.1	9.0	0.9	7.3	1.5
dopo la 2ª passata , in a abbiamo:	0.7	0.9	5.1	9.0	1.5	7.3	1.5
dopo la 3ª passata , in a abbiamo:	0.7	0.9	1.5	9.0	5.1	7.3	1.5
dopo la 4ª passata , in a abbiamo:	0.7	0.9	1.5	1.5	5.1	7.3	9.0
dopo la 5ª passata , in a abbiamo:	0.7	0.9	1.5	1.5	5.1	7.3	9.0
dopo la 6ª passata , in a abbiamo:	0.7	0.9	1.5	1.5	5.1	7.3	9.0

Note.

1. Nella 3ª passata, il valore minimo (tra quelli che consideriamo), cioè 1.5, è in due posizioni; potrei scegliere liberamente, ma bisogna darsi una regola per poter scrivere l'algoritmo; la regola scelta è quella di prendere il primo (quello più a sinistra).

2. Dopo la k-ma passata i primi k elementi dell'array (quelli sottolineati) sono "a posto" e non verranno più toccati.

3. Dopo la 4ª passata l'array è già ordinato, ma noi non lo sappiamo ... e il nostro PC, poverino, non è in grado di accorgersene; in effetti non ha la più pallida idea di cosa gli stiamo facendo fare

(questa è una banalità, ma è importante non dimenticarsene, soprattutto quando ci accorgiamo che il nostro programma non funziona e cerchiamo di capire perché). Quindi si continua; ci si ferma solo dopo n-1 passate (se l'array ha n elementi); infatti, in base alla nota 2, dopo n-1 passate, i primi n-1 elementi sono "in posizione definitiva", ma allora questo vale anche per l'ultimo.

Algoritmo

costanti: MAX = 100 *valore massimo per n (serve per fissare la dimensione di a)*

variabili: n : integer *per memorizzare l'input n*

a : array [1 .. MAX] of real

k, i, imin : integer *per indici ed altro ....*

istruzioni

leggi (n)

if (n > MAX) or (n < 1) then scrivi ("errore ....") *e si esce*

else {

memorizziamo la successione $a_1, ..., a_n$ nell'array; qui supponiamo che l'input sia corretto, cioè che ci siano (almeno) n numeri reali da leggere:

per k = 1, 2, ..., n : leggi(a[k])

riordiniamo l'array a: vedi sotto

stampiamo il risultato: per k = 1, 2, ..., n : scrivi(a[k])

}

Ordinamento dell'array

per k = 1, 2, ..., n-1 :

{

(1) trova la posizione più a sinistra che contiene il valore minimo tra quelli in a da k a n; chiamiamo imin l'indice di questa posizione;

(2) scambia a[k] con a[imin]

}

Nota.

Dopo aver eseguito (2) abbiamo: $a[1] \le a[2] \le ... \le a[k]$ ed inoltre

$a[k] \le a[p]$, per ogni p: $k < p \le n$

quindi a conclusione di tutto, quando k = n-1, abbiamo: $a[1] \le a[2] \le ... \le a[n]$ come si voleva.

Come si realizza (2).

E` semplice, basta una variabile ausiliaria, chiamiamola aux

aux ← a[k] ; a[k] ← a[imin] ; a[imin] ← aux

Come si realizza (1)

Servono due variabili intere : imin (vedi sopra) e j (per scorrere l'array).

Istruzioni:

imin ← k *il minimo dei valori dalla posizione k a quella in cui sono (sempre k) è in k*

per j = k+1, k+2, ..., n : if a[j] < a[imin] then imin ← j

a questo punto abbiamo: $k \le imin \le n$

$a[imin] \le a[p]$ *per ogni p,* $k \le p \le n$

$a[imin] < a[p]$ *per ogni p,* $k \le p < imin$

Rimettendo tutto assieme abbiamo la versione finale (per ora) dell'algoritmo.

Algoritmo

costanti: MAX = 100 *valore massimo per n (serve per fissare la dimensione di a)*

variabili: n : integer *per memorizzare l'input n*

a : array [1 .. MAX] of real

k, j, imin : integer *per indici ed altro ....*

aux : real *per effettuare scambio*

istruzioni:

leggi (n)

if (n > MAX) or (n < 1) then scrivi ("errore ....") *e si esce*

else {

memorizziamo la successione $a_1, ..., a_n$ nell'array; qui supponiamo che l'input sia corretto, cioè che ci siano (almeno) n numeri reali da leggere:

per k = 1, 2, ..., n **:** leggi(a[k])

riordiniamo l'array a:

per k = 1, 2, ..., n-1 **:**

{

trova la posizione più a sinistra che contiene il valore minimo tra quelli in a da k a n; chiamiamo imin l'indice di questa posizione:

imin ← k

per j = k+1, ..., n **:** if a[j] < a[imin] then imin ← j

scambia a[k] con a[imin]:

aux ← a[k] ; a[k] ← a[imin] ; a[imin] ← aux

}

stampiamo il risultato: per k = 1, 2, ..., n **:** scrivi(a[k])

}

10 . Array a più dimensioni

10.1 . Array a due dimensioni

Gli array a una dimensione corrispondono ai vettori; quelli a due dimensioni alle *matrici*.

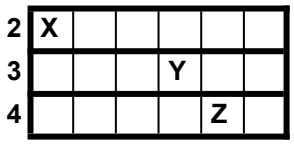
Una matrice a n righe e p colonne si può schematizzare come segue (qui n=4 e p=6):

indici di colonna →	**1**	**2**	**3**	**4**	**5**	**6**
1						
indici di riga → **2**	**X**					
3				**Y**		
4					**Z**	

Nella notazione matematica, se Mat è il nome della matrice, l'elemento **X** si indica con Mat(2,1), **Y** con Mat(3,4) e **Z** con Mat(4,5).

Però le cose si possono vedere anche in un altro modo:

	1	2	3	4	5	6
1						



La matrice Mat la vediamo come un vettore di 4 elementi (le righe), ciascuno dei quali è, a sua volta, un vettore di 6 elementi. Con questa visione: Mat(2) individua il vettore formato dalla seconda riga; il primo elemento di questo vettore, **X**, si indica allora, sempre seguendo la notazione matematica, con Mat(2)(1); analogamente: **Z** = Mat(4)(5) e **Y** = Mat(3)(4).

Nei linguaggi di programmazione gli array multidimensionali si introducono seguendo uno dei due modi visti. In Pascal l'approccio di base è il secondo, ma anche il primo è ammesso. Il C segue rigidamente il secondo approccio.

Vediamo come si dichiarerebbe e userebbe l'array Mat, supponendo che gli elementi siano "reali":

Pascal:

- dichiarazione **var** Mat **: array** [1..4] **of array** [1..6] **of** real oppure
  var Mat **: array** [1..4 , 1..6] **of** real
- uso: Mat [3][5] := 7.2 oppure Mat[3,5] := 7.2

In C gli indici partono da 0 e la dichiarazione è più concisa:

- dichiarazione: float Mat [4] [6]
- uso Mat[2][4] = 7.2

Notare che in C: **X** corrisponde a Mat[1][0], **Y** a Mat[2][3] e **Z** a Mat[3][4].

Noi useremo lo stile Pascal, nella versione più comoda (e più vicina allo stile matematico usuale).

Lo schema generale è:

- dichiarazione: **var** <nome> **:** **array** [<intervallo 1> , <intervallo 2>] **of** <tipo degli elementi>
- uso <nome> [<exp1> , <exp2>] **.....**

Esempio: (dichiarazione) **var** Tabella : **array** ['a' .. 'z" , 18 .. 33] **of** real
(uso) Tabella ['k', 22+3] ← 3.14

Un piccolo esempio di uso di array a due dimensioni: il problema del questionario.

Abbiamo un questionario con 20 domande, ciascuna con 4 alternative etichettate con a, b, c, d; al questionario hanno risposto 1000 persone; vogliamo un programma per sapere quante persone hanno scelto l'alternativa x nel rispondere alla domanda k.

Input: Il file RISPOSTE contiene 1000 righe della forma: r1 r2 r3 ... r20,
dove rj è una lettera in {a, b, c, d}, gli rj sono separati da uno spazio bianco;
quindi ogni riga del file contiene le risposte date da una persona.

Output: per ogni x in {a, b, c, d}, per ogni k in {1, ..., 20}:
il numero di persone che hanno risposto x alla domanda k.

Algoritmo

variabili somma : array [1..20, ‘a’ .. ‘d’] of integer

risposta : char

persona, domanda : integer

istruzioni:

inizializzazione della matrice somma (notare che usiamo delle istruzioni for annidate):

per domanda **=** 1, 2, ..., 20 **:**

per risposta **=** ‘a’, ‘b’, ‘c’, ‘d’ **:** somma[domanda, risposta] ← 0

lettura dell’input e aggiornamento della matrice:

per persona = 1, 2, ..., 1000 **:** {

per domanda **=** 1,2,....,20 **:** {

leggi(RISULTATI, risposta) *leggiamo dal file RISULTATI .....*

somma [domanda, risposta] ++ *vedi nota*

} *questa parentesi chiude l’istruzione “* ***per*** *domanda ....”*

passa alla riga successiva del file *(non dettagliamo questa parte)*

} *chiude l’istruzione “* ***per*** *persona ....”*

stampa i risultati *(non dettagliamo questa parte)*

Nota. somma [domanda, risposta] ++ è una abbreviazione, nello stile C; equivale a:

somma [domanda, risposta] ← somma [domanda, risposta] + 1

Esercizi. Scrivere gli algoritmi per la moltiplicazione di due matrici e per produrre la trasposta di una matrice.

10.2 . Array multidimensionali generali

Generalizzazione a k dimensioni di quanto visto per quelli a due dimensioni.

Ad esempio, per k=3:

in Pascal:

dichiarazione **var** Mat **: array** [1..4] **of array** [1..6] **of array** [1..7] **of** real oppure

var Mat **: array** [1..4 , 1..6 , 1..7] **of** real

uso: Mat [3][5][7] := 7.2 oppure Mat[3,5,7] := 7.2

in C:

dichiarazione: float Mat [4] [6] [7]

uso Mat[2][4][6] = 7.2

11 . I record

Negli array, le componenti, o elementi, sono omogenee. Se consideriamo una scheda anagrafica per una persona, abbiamo componenti che sono disomogenee: nome e cognome sono stringhe, anno di nascita è un numero ...

Per permettere di gestire facilmente questo genere di situazioni, molti linguaggi di programmazione offrono i *record* (in C si usa la parola *structure*, abbreviata in `struct` , ma la sostanza non cambia).

Consideriamo schede anagrafiche molto semplici: nome, cognome, anno di nascita, vivente?
dove:

nome e cognome sono stringhe;

anno di nascita è un intero (se negativo, indica nascita avanti Cristo)

vivente? è un booleano

Quindi le schede hanno 4 *campi*; ogni campo ha un nome o *etichetta* (nome, cognome,...) ed un *tipo* (stringa, intero,...) che determina quali valori può assumere (o contenere) quel campo.

Usando una sintassi di tipo Pascal, la *definizione* che corrisponde alle nostre schede ha la forma seguente, abbreviando un po' le etichette dei campi e fissando una lunghezza massima (const MAX = ...) per le stringhe :

```
record    nome : array [1 .. MAX ] of char
          cogn : array [1 .. MAX ] of char
          anno : integer
          vive : boolean
end
```

Questa definizione si usa in due modi: in una dichiarazione di tipo (che vedremo tra poco), o in una dichiarazione di variabile. Ad esempio, la dichiarazione:

```
var x, y : record    nome : array [1 .. MAX ] of char
           cogn : array [1 .. MAX ] of char
           anno : integer
           vive : boolean
           end
```

dice che x ed y sono due identificatori di variabile che assumeranno valori del nostro tipo record.

Più concretamente: ad x si assocerà l'indirizzo di una "cella" Cx ed analogamente per y.

La cella Cx è strutturata in 4 parti: le prime due sono due array di caratteri, la terza è una "cella per intero" e la quarta una "cella per booleano". Ciascuna di questa quattro parti può essere *selezionata*, usando i seguenti nomi (identificatori strutturati):

x.nome x.cogn x.anno x.vive

quindi della forma:

<nome della variabile record> . <etichetta di campo>

Questi nomi sono, a tutti gli effetti, *nomi di variabili*; quindi, ad esempio, si può scrivere:

```
x.anno ← 1567
y.anno ← x.anno + 10
per k = 1, 2, ..., MAX : leggi ( x.cogn[k] )
```

A livello di pseudo-codice, scriveremo anche cose come

y ← x oppure leggi(x)

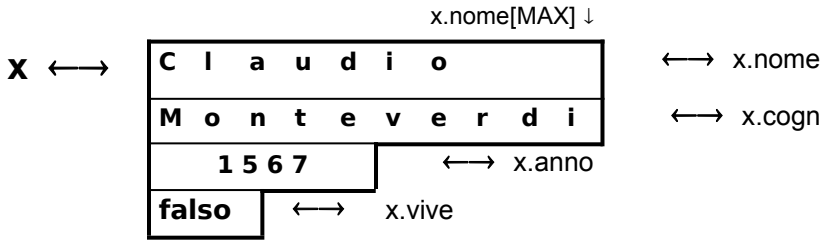
ma deve essere chiaro che queste pseudo-istruzioni corrispondono, in genere, ad una successione di istruzioni in un linguaggio di programmazione "vero". Quello che viene garantito nei linguaggi di programmazione che prevedono i record è di poter selezionare le componenti elementari, cioè

corrispondenti ad un tipo predefinito (intero, carattere,...) e di operare su di esse come se si trattasse di una normale variabile di quel tipo. Nel nostro esempio:

x.cogn può essere usata come una qualunque altra variabile di tipo array

x.cogn[k] può essere usata come una qualunque altra variabile di tipo carattere, eccetera.

Graficamente possiamo rappresentare la situazione così (con MAX = 10):



Record con varianti

Presenti in Pascal, ed in altri linguaggi, ma non in C. Verranno visti parlando delle *union* (tipi unione, corrispondenti al concetto matematico di unione disgiunta) del C.

Record in C. Ne parleremo brevemente nella sezione dedicata alle dichiarazioni di tipo.

Per vedere un altro esempio dell'uso dei record, consideriamo i numeri complessi. Questi sono coppie di reali, dunque si possono facilmente rappresentare come array di due elementi; tuttavia è più chiaro se li rappresentiamo con dei record a due campi:

```
record    re : real    la parte reale
          im : real    la parte immaginaria
end
```

Infine i record sono utilizzati, assieme ai puntatori per realizzare *strutture dinamiche* (liste, alberi,....) come vedremo.

12. Tipi e dichiarazioni di tipo

Abbiamo visto che un linguaggio di programmazione offre:

- tipi di base, predefiniti, corrispondenti, in qualche modo, a: interi, razionali, caratteri,.... e per questi fornisce direttamente le costanti e le operazioni principali;
- strumenti per costruire tipi complessi, strutturati.

Relativamente a questi ultimi, abbiamo visto gli array e i record (ma vedere precisazioni più avanti). Alcuni linguaggi forniscono molto di più, magari introducendo i concetti di oggetto e classe (ad esempio, C++ e Java); noi ci limiteremo a quello che offre il C e quindi vedremo ancora i file (non in queste note), le union e i puntatori.

Prima di andare avanti è opportuno introdurre le **dichiarazioni di tipo**. Cominciamo con il Pascal dove hanno un ruolo centrale.

Abbiamo visto dichiarazioni di variabile della forma:

```
var  A, B  : array [1 .. MAX]  of  integer
var  x, y :   record      nome : array [1 .. MAX ] of char
                          cogn : array [1 .. MAX ] of char
                          anno : integer
                          vive : boolean
              end
```

Queste dichiarazioni definiscono le variabili A, B, x, y, *precisandone la* ***struttura***.

È possibile definire le strutture stesse. Nello stile Pascal:

```
type  Vettore = array [1 .. MAX]  of  integer
type  Scheda = record      nome : array [1 .. MAX ] of char
                           cogn : array [1 .. MAX ] of char
                           anno : integer
                           vive : boolean
              end
```

Questi sono esempi di dichiarazioni di tipo (e si riconoscono per la parola chiave **type**). Dal nostro punto di vista, il loro ruolo è quello di poter definire, a parte, una struttura ed un nome per essa. Dopo queste dichiarazioni, invece di scrivere **array [**1 .. MAX**] of** integer basta scrivere Vettore.

In particolare, invece di **var** A, B : **array [**1 .. MAX**] of** integer

basta scrivere **var** A, B : Vettore

Per capire l'utilità, vediamo un altro modo di organizzare le dichiarazioni per le nostre schede:

```
const  MAX = ....
type   Stringa = array [1 .. MAX] of char
       Scheda =  record      nome : Stringa
                             cogn : Stringa
                             anno : integer
                             vive : boolean
                 end
var    x, y : Scheda
       s1, s2 : Stringa
```

Nota. In Pascal le dichiarazioni si raggruppano per categoria: prima tutte quelle di costante, poi tutte quelle di tipo, poi tutte quelle di variabile; ogni gruppo è preceduto dalle parole chiave **const**, **type**, **var**; in C c'è più libertà, e quindi più confusione.

Si dovrebbe capire il vantaggio di una simile organizzazione: la definizione della struttura che serve per le stringhe si dà una volta sola, si vede subito che s1, s2 sono dello stesso tipo di x.nome,.....

Quando parleremo di funzioni e procedure, si vedrà un altro motivo: nella dichiarazione di una funzione o procedura, bisogna specificare il tipo dei parametri, o argomenti; nel caso di argomenti che sono array o record è comodo definire prima il tipo e dargli un nome e poi, nella dichiarazione di procedura usare solo il nome (il Pascal, addirittura, costringe a fare così).

Tipi e costruttori di tipo.

C'è un punto da chiarire (per quanto possibile al momento). Consideriamo le dichiarazioni:

type Stringa **= array [**1 .. MAX**] of** char

var s1, s2 : Stringa

Qui:

- array [1 .. MAX] of char è la *descrizione di un tipo:* dice come sono fatti gli oggetti che vogliamo usare: vettori di caratteri di lunghezza MAX (e quali operazioni posso applicare ad essi);
- Stringa è il nome che abbiamo dato a questo tipo;
- s1, s2 sono nomi di variabili di tipo Stringa; ad essi corrispondono valori destri che sono oggetti di tipo Stringa; ad essi possiamo applicare le operazioni previste; in Pascal standard solo una: *selezionare una componente* (questo si fa scrivendo ad esempio: s2[1]);
- array è un *costruttore di tipi*: dato un tipo per gli indici ed uno per gli elementi restituisce il tipo degli array con quegli indici e quegli elementi; quindi non è corretto dire "**il** tip**o** array", perché non esiste; esistono **tanti** "tip**i** array".

Discorso analogo vale per record e unioni.

12.1 . Breve rassegna sui tipi *user-defined*

Le definizioni di tipo non sono limitate al caso degli array e dei record. Vediamo brevemente le più frequenti (non vediamo tutte le possibilità!) presentando la versione Pascal e, se esiste, quella C, dove le dichiarazioni di tipo hanno la forma:

typedef <descrizione del tipo> <nome del tipo>;

Tipi enumerazione.

Un esempio Pascal: la dichiarazione **type** Luci **=** (VERDE, ROSSO, GIALLO)

definisce un tipo, Luci (le luci di un semaforo) con 3 oggetti, chiamati VERDE, ROSSO, GIALLO; come operazioni possibili: l'uguaglianza, l'operazione di successore (succ(VERDE) = ROSSO), ...

In C si scriverebbe **typedef enum** {VERDE, ROSSO, GIALLO} Luci;

poi la dichiarazione Luci luce1, luce2; dichiara due variabili di tipo Luci.

Non è tutto qui, ma in quello che segue non useremo i tipi enumerazione, quindi può bastare.

Tipi intervallo

Presenti in Pascal, non in C. In Pascal, scrivendo

- **var** k : 10 .. 70 si dichiara che k è una variabile di tipo intero che però può assumere valori solo nell'intervallo da 10 a 70 (estremi compresi)
- **var** ch : 'D' .. 'P' si dichiara che ch è una variabile di tipo carattere che può assumere valori solo nell'intervallo da 'D' a 'P' (estremi compresi)
- **type** Indice **=** 1 .. MAX si definisce un tipo intervallo (da 1 alla costante MAX) e lo si chiama Indice; poi si può ad esempio scrivere Vettore = array [Indice] of integer
- **type** Nat = 0 .. MAXINT fornisce una approssimazione al tipo *naturali*; MAXINT è una costante predefinita, e dipendente dal sistema, che fornisce il valore massimo per gli interi.

Per i tipi intervallo vale quanto detto a proposito degli indici degli array: l'insieme dei valori di un tipo intervallo deve poter essere messo in corrispondenza biunivoca con l'insieme dei naturali da 0 a k, per un qualche k (questo collegamento non è casuale: gli indici di un array devono proprio essere di tipo intervallo).

Sinonimi

Vediamone gli usi più frequenti in C.

1. Non c'è il tipo booleano e quindi una variabile, sia b, che a livello di algoritmo è booleana, quando l'algoritmo viene tradotto in C, va dichiarata con un tipo intero.

Se però dichiariamo **typedef** int Boolean ;

Boolean diventa un sinonimo di " int " , quindi se dichiariamo Boolean b ;

la variabile b è di tipo intero a tutti gli effetti, però si capisce che concettualmente è di tipo booleano.

(Bisogna dire, però, che alcuni ambienti di programmazione per il C offrono un tipo Boolean e le costanti TRUE, FALSE; si arrabbiano, quindi, se proviamo a ridefinirli.)

1bis. Poiché i char non sono altro che degli interi piccoli e per i booleani servono solo due valori, si può anche dichiarare **typedef** char Boolean ;

2. Il C offre diversi tipi interi: normali, lunghi, corti, cortissimi (i char); se abbiamo qualche dubbio su quale di queste varianti vada meglio in un certo programma, possiamo o dichiarare

typedef int Interi;

all'inizio del programma e poi usare Interi in tutti i punti in cui avremmo usato int (o long int, o....).

Se poi gli int non bastano e ci vogliono i long int basta cambiare la riga del typedef.

Analogamente per il caso dei razionali (float, double) o dei naturali (ottenibili come unsigned int, unsigned long int, ...).

Tipi array

Per il Pascal non aggiungiamo nulla a quanto già visto. Per il C: non c'è un equivalente per una dichiarazione come

type Vettore = array [1 .. MAX] of integer

ma solo per una dichiarazione come

var a : array [1 .. MAX] of integer (nella forma: int a[MAX]).

In altre parole in C non esistono i tipi array in quanto tali; quando si scrivono programmini come facciamo noi questo aspetto non è rilevante.

Tipi record

Per il Pascal non aggiungiamo nulla a quanto già visto. Per il C l'esempio interessante, tra quelli visti, è:

```
type  Scheda = record      nome : array [1 .. MAX ] of char
                               cogn : array [1 .. MAX ] of char
                               anno : integer
                               vive : boolean
                      end
```

che in C si scrive (notare la parola chiave **struct**):

```
typedef  struct    {   char  nome [MAX] ;
                       char  cogn [MAX] ;
                       int anno;
                       int  vive;
                   }   Scheda
```

oppure (notare che bisogna ripetere la parola **typedef**):

```
typedef  char Boolean;
typedef  struct    {   char  nome [MAX] ;
                       char  cogn [MAX] ;
                       int anno;
                       Boolean  vive;
                   }   Scheda
```

Quando poi si vogliono dichiarare delle variabili si scrive, ad esempio: Scheda x, y ;

Nota: in C ci sono tanti modi di dichiarare le struct e le variabili di tipo struct; qui abbiamo presentato solo una possibilità (quella che suggeriamo).

Altri tipi di tipi.

Restano da vedere: tipi union e file (non in queste note) e tipi puntatore.

12.2 . Controlli sui tipi

C'è una differenza notevole tra Pascal e C per quanto riguarda il controllo sul rispetto dei tipi. La filosofia di base del Pascal è che i tipi sono tutti diversi ed incompatibili, tranne qualche eccezione.

Supponiamo di aver dichiarato in Pascal

```
var     c : char ;   n : integer ;   x : real ;
        k : 1 .. 100;        (cioè  k  è una variabile di tipo intervallo)
```

Allora è vietato scrivere, ad esempio:

c := 3 oppure n := 'a' oppure n := c+1

Il compilatore del Pascal cattura tutti i tentativi di mescolare i tipi.

Le uniche eccezioni riguardano un po' di conversioni automatiche tra interi e razionali (x := n+1 è accettato) e tra tipo base e suo intervallo (n:= k va sempre bene; k := n è controllato a run time, per vedere se il valore di n è accettabile).

In particolare, i tipi `integer`, `char` e `boolean` sono incompatibili (non mescolabili).

In C, invece i tipi `int` e `char` sono perfettamente compatibili; infatti il tipo char è essenzialmente un tipo intero: quello degli interi rappresentabili con un byte.

Quindi dopo aver dichiarato char c ; int n ;

nulla vieta di scrivere: c = 3; n= 'a' ; printf(.... , n+c).

Per quanto riguarda i problemi di compatibilità con gli array, C e Pascal non sono confrontabili perchè in C non esistono tipi array. Il risultato è che il C offre più flessibilità (vedere oltre: array e puntatori, array come parametri di funzioni e procedure), ma meno assistenza in caso di errori.

I due linguaggi sono invece molto simili nel modo di gestire la compatibilità tra i tipi record. Per capirlo, vediamo un esempio con la sintassi C:

```
struct  { int a ;  float b; } x, y;
        /* x  e  y  sono due variabili di tipo record a 2 campi ... */
struct  { int a ;  float b; } z ;
        /* anche  z  e` un  record a due campi,.... nella sostanza,
           x, y , z  sono dello stesso tipo, pero` .....          */
x.a = 3;
x.b = 5.0;
y = x;          /* questa assegnazione va bene      */
z = x;          /* questa viene rifiutata dal compilatore !   */
```

Il punto è che sia il compilatore C che quello del Pascal si comportano in modo molto burocratico, attaccati alla forma e non alla sostanza; quindi due definizioni di tipo distinte vengono considerate comunque diverse (il punto è che, in esempi più complicati, è molto difficile, se non impossibile, star dietro alla sostanza; allora si è preferita la soluzione drastica).

Quindi il controllo dei tipi in C è inesistete in certi casi, fin troppo rigido in altri; comunque offre molte possibilità di fare pasticci e sta al programmatore evitarli.

Un modo di evitare i pasticci, quando si sviluppa un programma partendo “da zero”, è:

- progettare l’algoritmo rimanendo vicini al problema e facendo riferimento ai tipi del problema (booleani, naturali, intervalli, luci di semaforo, successioni, stringhe, schede) arrivando fino ad un algoritmo completo e dettagliato; a questo livello, è difficile che si provi a sommare un carattere con un booleano;
- usare nomi significativi e annotare, con commenti, il ruolo delle singole variabili; così si riduce il rischio di confondere una variabile di tipo carattere con una di tipo stringa (array di caratteri) e cose del genere;
- fare le codifiche necessarie (booleani ↔ char, true ↔ 1, false ↔ 0, giallo ↔ 2, ...) solo all’ultimo, quando si tratta di tradurre l’algoritmo in C; inoltre usando i #define ed i typedef è anche possibile usare TRUE, FALSE, GIALLO, Booleani, ... nel programma C (notare che i #define ed i typedef NON penalizzano in fase di esecuzione).

13. Puntatori ed indirizzi

In C ed in Pascal (ed in altri linguaggi) esistono i tipi puntatore.

I *valori* sono *indirizzi* (gli *indirizzi di memoria* visti ad Architettura degli elaboratori); quindi, in sostanza, degli interi non negativi.

Le *operazioni* dipendono dal linguaggio: ci concentreremo sul C (più ricco in questo del Pascal).

Se T è un tipo (di base o strutturato), si può definire un tipo: puntatore_a_T.

Quindi esistono tanti tipi puntatori diversi; ma i valori e le operazioni sono sempre le stesse. Questo sconcerta un po'; in effetti, quello che cambia, in base al tipo T, è l'effetto di alcune operazioni (e di alcune funzioni, o procedure, pre-definite) come vedremo.

Dichiarazioni di variabili di tipo puntatore.

Forma in Pascal: var p1, p2 : **^**T

in C: T ***p1, *p2**

noi: p1, p2 : puntatori_a_T

dove T può essere un nome di tipo (predefinito, come int, oppure definito dal programmatore, come Scheda) o anche la descrizione esplicita di un tipo, come struct {....}).

Sostanza del tutto analogo a quanto visto parlando di dichiarazioni di variabile:

a p1 e p2 si associano due celle, ciascuna buona per contenere un indirizzo.

Si possono fare anche **dichiarazioni di tipo puntatore**, ad esempio:

```
typedef  struct { .....}  Scheda ;        /* un tipo record */
typedef  Scheda *  Pun_a_Scheda ; /* il tipo dei puntatori a questi record  */
```

Quando si ha una variabile, bisogna poter assegnarle dei valori, lavorarci sopra,.....

Nel caso dei puntatori abbiamo visto che i valori sono indirizzi, quindi si potrebbe pensare, avendo una variabile p di tipo puntatore_a_... di fare delle assegnazioni come p ← 1024 , cioè assegnare a p l'indirizzo 1024. Però per fare cose del genere senza combinare pasticci, bisognerebbe sapere dove viene caricato in memoria il nostro programma, avere la mappa completa di quello che c'è in memoria, ...; quindi si deve seguire un'altra strada.

Vediamo **come ottenere dei valori indirizzo** (cioè delle costanti di tipo puntatore) utili.

- l'*indirizzo nullo*, indicato con nil in Pascal e con NULL in C, è una costante di tipo puntatore_a_T, per qualunque T; il suo ruolo si capirà parlando di liste ed alberi.
- in C (non in Pascal), se x è un nome di variabile, &x è l'indirizzo di x (quindi & indica l'operazione che preso un nome di variabile ci fornisce l'indirizzo della variabile, cioè della cella); notare che &x è una costante, non una variabile;
- in C (non in Pascal): dopo aver dichiarato:
  T aa[10] (cioè: aa è un array con elementi di tipo T)
  aa è una costante di tipo puntatore_a_T ; il suo valore è l'indirizzo di aa[0] ;
  questo legame stretto tra array e puntatori fornisce al C molta flessibilità (es: array dinamici), ma è anche fonte di confusione e di pasticci; vedremo i vari aspetti un po' alla volta;
- infine ci sono delle *funzioni / procedure di libreria* (new, malloc, calloc, ...) che vedremo più avanti.

Notare che dopo aver dichiarato: T aa[10] si ha:

aa	ha come valore l'indirizzo di aa[0]	
&aa[0]	ha come valore l'indirizzo di aa[0];	notare che &aa[0] = &(aa[0])
&aa[1]	ha come valore l'indirizzo di aa[1]	eccetera

Analogamente, dopo le dichiarazioni

```
typedef  struct      {   char  nome [10] ;
                         char  cogn [10] ;
                         int anno;
                         int  vive;
                     }   Scheda ;
Scheda   sss ;
```

abbiamo che

&sss	ha come valore l'indirizzo di tutto il record sss
	(qui NON c'è analogia con il caso di aa)
sss.cogn	ha come valore l'indirizzo di sss.cogn[0]
&sss.cogn[3]	ha come valore l'indirizzo di sss.cogn[3]
&sss.anno	ha come valore l'indirizzo di sss.anno

Nel caso degli array (e solo per essi) , il C fornisce quella che è detta "aritmetica dei puntatori".
Dopo aver dichiarato: T aa[10]

aa	è l'indirizzo di aa[0]	
aa+1	è l'indirizzo di aa[1];	quindi: aa+1 = &aa[1]
aa+<exp>	è l'indirizzo di aa[k] se k è il valore di exp	

Se p è una variabile di tipo puntatore_a_T (e p ha un valore) allora è possibile **accedere alla cella puntata** da p, scrivendo:

^p in Pascal e ***p** in C.

Ora si può capire a che serve conoscere il tipo T; infatti il valore di p è un indirizzo, ad esempio 1024; se T è char, l'implementazione sa che la cella che ci interessa è il byte all'indirizzo 1024; se T è int, e gli int sono lunghi 2 byte, allora la cella che ci interessa è formata da due byte, a partire da 1024; eccetera. (Anche per l'aritmetica dei puntatori, serve sapere il tipo degli elementi dell'array).

Per capire veramente le cose, ci vorrebbe un esempio in cui effettivamente i puntatori servono; purtroppo è ancora presto per farlo, quindi ci accontentiamo di un esempio artificiale (e quindi stupido).

```
#include <stdio.h>

 main( )

{   int x;
   int * p, * q ;   /*   p e q  sono variabili   puntatore_a_int */

   p = &x;  /* x non ha ancora un valore (in realta` ne ha uno "default")
            ma ha gia` un indirizzo.
            A questo punto la cella per x si puo` raggiungere sia usando il
            nome x, che attraverso il puntatore p, scrivendo:  *p
          */
   q = p;  /*  ora anche q punta alla cella per x  */

   *q = 3;

   printf (" %d \n", x);      /*   output 3  */

   x = x+1;

   printf (" %d \n", *p);     /*   output 4  */
}
```

14. Ancora su variabili e assegnazione

Ritorniamo sulla questione “variabili” per qualche precisazione, riferendoci soprattutto al C.

Il punto è che le dichiarazioni:

```
int  n, aa[5] ;
typedef struct  { float  x, y ; }  Punto ;
Punto  centro;
```

introducono (oltre al tipo Punto):

- una variabile semplice con nome semplice: n
- una costante di nome aa (infatti, in C, aa non è un nome di variabile)
- cinque variabili semplici con nomi strutturati (o composti): aa[0], aa[1], ..., aa[4]
- una variabile strutturata (o composta) con nome semplice: centro
- due variabili semplici con nomi strutturati: centro.x e centro.y

Con i puntatori poi le cose si complicano ulteriormente; con aa come sopra: *(aa + k) equivale a aa[k], quindi ....

Il primo punto è: **la distinzione tra variabili e nomi di variabile**. Spesso, per brevità si confonde nome con varabile, ma è importante saper fare la distinzione.

Abbiamo visto che l’effetto della dichiarazione int n ; è quello di

- riservare un “cella per intero”, sia Cn; questa cella è la variabile;
- associare un nome, cioè n, a questa cella,.... eccetera.

Ricordiamo che una cella non è un indirizzo, un byte, una “parola”, o altro, nella macchina fisica, ma è un qualcosa a livello di “macchina virtuale C”, per cui non c’è solo la garanzia che la dimensione sia giusta, ma anche la possibilità di qualche controllo sui valori che può contenere.

La dichiarazione Punto centro ;

definisce tre variabili: la cella che contiene tutto il record (con nome centro) e due "sottovariabili", le due celle che corrispondono ai due campi (con nome: centro.x e centro.y). La variabile di nome centro è un esempio di variabile strutturata.

Variabili semplici e strutturate.

Le variabili che sono formate da varie parti, che a loro volta, sono variabili, sono dette *variabili strutturate* (o composte). Per contrasto, le variabili per i tipi base (interi, caratteri, razionali, puntatori) sono dette *variabili semplici*; infatti non contengono parti che sono delle variabili.

Le variabili di tipo record sono strutturate, sia in Pascal che in C.

Per gli array Pascal e C divergono.

In Pascal la dichiarazione var bb : array [....] of T definisce una variabile strutturata, che corrisponde a tutto l'array, con nome bb, e tante variabili di tipo T, con nome bb[...].

In C, la dichiarazione T bb [MAX] definisce solo tante variabili di tipo T, con nomi: bb[0], ... bb[MAX-1], mentre non c'è nessuna variabile che corrisponde a tutto l'array; bb non è un nome di variabile, ma un nome di costante, di tipo puntatore_a_T, il cui valore è l'indirizzo di bb[0].

Nelle sezioni precedenti, in qualche punto, abbiamo sorvolato su questo e parlato di variabili di tipo array anche riferendoci al C. In effetti, per quello che stavamo facendo, era un'approssimazione accettabile.

Vedremo che anche le variabili di tipo unione sono strutturate.

Nomi semplici e strutturati; nomi che sono espressioni.

I nomi semplici sono identificatori. Quelli strutturati (o composti) sono quelli che identificano:

- gli elementi di un array (cioè le variabili che corrispondono ai singoli elementi); la forma tipica è: <nome dell'array>[<espressione>]
- i campi di un record (cioè le variabili corrispondenti ai campi); forma tipica: <nome del record>**.**<nome del campo>
- le alternative di una unione (vedere oltre).

Infine ci sono nomi che sono *espressioni di tipo puntatore*, ad esempio:

*(&n) *(aa + 1) *(aa + fun(...)) dove fun è una funzione ....

Notare che una variabile, almeno in C, ha più nomi; ad esempio : n e *(&n); aa[1] e *(aa +1).

Nel primo caso si può dire che n è il nome principale, quello attribuito dalla dichiarazione, nel secondo caso la situazione è dubbia.

Un modo per uscire da questa confusione, sarebbe quello di definire un tipo particolare di espressioni, che potremmo chiamare "l-espressioni", o "espressioni sinistre", di cui quelli di sopra siano tutti casi particolari, e non parlare più di "nomi" ma di l-espressioni.

Assegnazione

La forma generale dell'assegnazione in C (trascurando le varianti con +=, *=,....) è quindi:

<nome di variabile> = <espressione> ;

dove <nome di variabile> può essere complicato a piacere, come visto.

Variabili statiche e dinamiche.

Ne parleremo dopo aver parlato di procedure e di visibilità delle dichiarazioni.

15. Le union del C

I tipi record corrispondono al concetto matematico di prodotto cartesiano: se dichiariamo

```
typedef struct { <tipo_1> <nome_campo_1> ;
                 <tipo_2> <nome_campo_2> ;
                 ..............................
                 <tipo_k> <nome_campo_k> ;
               } TTrr ;
```

e Vj è l'insieme dei valori per il tipo <tipo_j>, allora l'insieme dei valori per il tipo TTrr è V1 × V2 × .... × Vk .

I tipi unione, corrispondono, invece, al concetto matematico di unione insiemistica (per essere precisi: *unione disgiunta*). Dichiarando

```
typedef union { <tipo_1> <nome_alternativa_1> ;
                <tipo_2> <nome_alternativa_2> ;
                ..............................
                <tipo_k> <nome_alternativa_k> ;
              } TTuu ;
```

l'insieme dei valori per il tipo TTuu è, con Vj come sopra: V1 ∪ V2 ∪ .... ∪ Vk .

Una variabile yy di tipo TTuu può avere valori in uno qualunque degli insiemi Vj; quindi i tipi <tipo_j> sono in alternativa. Se v2 e vk sono valori, rispettivamente, in V2 e Vk, si scrive

```
yy.<nome_alternativa_2> = v2          per assegnare ad yy il valore v2
yy.<nome_alternativa_k> = vk              per cambiarlo in vk
```

L'implementazione del C si limita a dimensionare la cella per yy in modo che possa contenere qualunque valore tra quelli possibili; è compito del programmatore ricordarsi, in ogni momento, quale tipo di valore contiene la cella in questione.

Vediamo un esempio concreto:

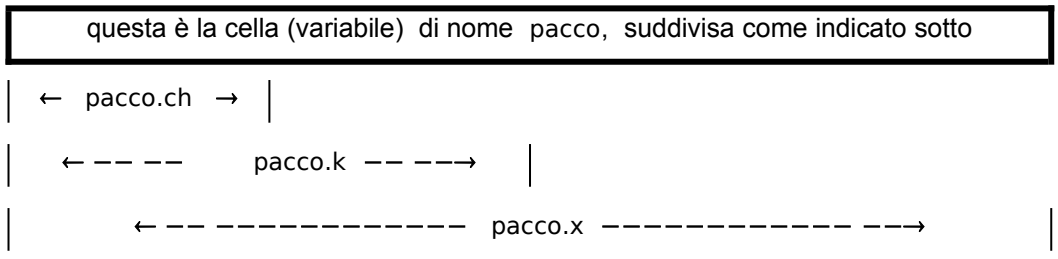
```
typedef union { char ch ; int k ; float x } Pacco ;
Pacco pacco;
```

Qui abbiamo definito il tipo Pacco e 4 variabili:

una variabile di tipo Pacco e di nome pacco

tre sottovariabili, di tipo char, int e float e con nome: pacco.ch, pacco.k e pacco.x.

Schematicamente:



A livello di macchina fisica, la variabile pacco e la variabile pacco.x occupano esattamente lo stesso spazio di memoria; a livello di macchina astratta, le due variabili sono diverse perché hanno un tipo diverso. Quindi (con il mio compilatore):

```
pacco.x = 3.14;      viene accettato, mentre
pacco = 3.14;        provoca un errore:  cannot convert 'float' to 'union'
```

L'esempio più semplice di uso dei tipi unione si ha quando si vogliono definire record che hanno una parte "fissa" ed una "variabile". Pensiamo, ad esempio, a dei record per contenere informazioni su barche a remi, a vela (1 solo albero) ed a motore. Questi record potrebbero avere

parte fissa (cioè per tutti i tipi di barca):		lunghezza (in m) : razionale
		larghezza max (in m) : razionale
		eccetera
parte variabile:	se è a remi:	numero di remi : intero
	se è a vela:	altezza albero (in m) : razionale
		superficie vela (in mq) : razionale
	se è a motore:	entrobordo? : booleano
		potenza motore (in kW) : intero

Quindi, la parte variabile è: un intero, oppure una coppia di razionali, oppure una coppia (booleano, intero); queste tre alternative sono mutuamente esclusive, quindi è comodo rappresentarle con un unico campo di tipo unione; in più serve un campo (che_barca) per distinguere i tre casi.

In C tutto questo si può scrivere così:

```
typedef  enum { REMI, VELA, MOTORE } Gen_barca ;

typedef  union  {   int   se_remi ;

              struct { float  h_alb;
                     float  sup_vela ;
                   }   se_vela ;

              struct { int  entro ;
                     int  pot ;
                   }   se_motore ;

           }   RemiVelaMotore ;

typedef  struct  {  float  lung ;
              float  larg ;
              .................
              Gen_barca  che_barca;
              RemiVelaMotore  spec ;

            }  Barca ;
.....................................

Barca x , y;

x.lung = 3.14 ;
x.larg = 1.0  ;
x.che_barca = REMI;
x.spec.se_remi = 2 ;     /*  abbiamo costruito una barca a remi  */

y = x ;              /*  salviamo la barca  prima di modificarla  */

x.che_barca = VELA;
x.spec.se_vela.h_alb = 2.5 ;
x.spec.se_vela.sup_vela = 9.5;    /*  ora  x è a vela ....  */
.............................
```

16. Procedure e funzioni

Un programma in linguaggio macchina è una lista di istruzioni; ciascuna istruzione è molto semplice, ma, in genere, il programma è incomprensibile. In effetti, una lista di 10 istruzioni semplici può ancora essere semplice da capire, ma una lista di 1000 non lo è quasi mai.

Per avere chiarezza, servono (almeno): *sintesi* e *struttura*; questa è la strada seguita, almeno in linea di principio, dai linguaggi evoluti moderni, che offrono schemi di base orientati a questo scopo, ad esempio: if-then-else, while,...

Però questi schemi non bastano; allora entrano in gioco procedure e funzioni (versione ad alto livello delle *subroutines* presenti nei linguaggi assembler).

Ci sono diverse cose da dire a proposito di procedure e funzioni. Qui cerchiamo di dare un'idea e precisiamo alcuni punti.

16.1 . Un esempio di funzione

Riprendiamo il problema dei numeri primi (visto all'inizio). Lo schema di soluzione visto era:

- 1 e 2 sono primi
- per n = 3, 4, ..., 100: **controllo se n è primo**
  se è primo, lo stampo

La parte **controllo se n è primo** si dettagliava nel modo seguente:

controllo per k = 2, 3,...., fino a circa la radice quadrata di n se k divide n .....

Volendo mantenere anche nel programma questo modo di scrivere le cose, la parte **controllo se n è primo** si formula come una procedura o una funzione.

Poiché tutto quello che vogliamo sapere dal controllo è: vero/falso, una funzione è la cosa migliore.

Si tratta di definire una funzione primo : Interi → {vero, falso} tale che

primo(x) = vero se x è primo
falso altrimenti

Per come abbiamo fatto le cose, in realtà serve primo3 : {x | x intero ≥3} → {vero, falso} tale che .....

In matematica, a livello elementare:

- prima si definisce una funzione, specificando:
  1. nome, dominio e codominio (esempio f: A → B)
  2. in qualche modo, qual'è, o come si ottiene, il valore della funzione, per un generico argomento x (esempio f(x) = x+3)
- poi, si usa la funzione, applicandola ad argomenti specifici, all'interno di espressioni (es: 1+ f(4)/2)

Nei linguaggi di programmazione la cosa è analoga: le funzioni si definiscono con una *dichiarazione di funzione* e poi si usano all'interno di *espressioni*.

Tornando al nostro esempio, vediamo una *dichiarazione* di primo3 in stile Pascal:

La riga (1) è *l'intestazione* e specifica: il nome della funzione (primo3), il nome del *parametro formale* (o argomento generico, o variabile, come si dice in analisi, che è x), il dominio (gli interi), il codominio (i booleani).

In generale, si possono considerare funzioni con più parametri, o argomenti, proprio come in matematica.

La riga (2) contiene *dichiarazioni interne* alla funzione, cioè di variabili (o, in genere, altro) che vengono usate solo all'interno della funzione (chiariremo meglio parlando di Visibilità delle dichiarazioni).

A partire dalla riga (3), c'è il *corpo della funzione*, cioè la parte istruzioni, il cui compito è definire come si calcola il valore della funzione sull'argomento x.

```
(1) function primo3 (x : integer) : boolean;
            nota: assumiamo x ≥ 3
(2)      var  k : integer;    ancora : boolean;

(3)      begin
         k := 2;
                  ancora := true;
         while (k*k <= x) and ancora do
           begin    if (x mod k) = 0  then ancora := false;
                  k := k+1
           end ;
(4)          if  ancora  then   return(true)  else return(false)
         end
```

Note.

1. Nella riga (4) compare l'istruzione **return (<espressione>)**. E` una istruzione che si usa per dire qual'è il valore che deve essere "restituito" al programma che ha chiamato la funzione. Questo si chiarirà meglio dopo aver messo tutti i pezzi assieme.

2. La riga (4) si poteva scrivere (meglio) come: `return(ancora)`

3. In realtà in Pascal, non si scrive " return(exp) ", ma " primo3 := exp " ; in effetti questo è uno degli aspetti "brutti" del Pascal.

Nel programma, usiamo la funzione primo3 all'interno di espressioni, ad esempio:

```
if primo3(n) then writeln (n, " e` primo")
```

Notare che qui l'espressione è la condizione dell'istruzione if-then-else e si riduce alla sola *chiamata di funzione*.

Infine, in Pascal la dichiarazione della funzione compare, nel programma assieme alle altre dichiarazioni ed in testa al programma.

Programma in Pascal completo (e seguendo, salvo errori, la sintassi Pascal):

```
program  primi(input, output);

 const  MAXNUM = 100;
 var  n : integer;

 function primo3 (x : integer) : boolean;
        var  k : integer; ancora : boolean;
        begin
      k := 2;
            ancora := true;
      while (k*k <= x) and ancora do
          begin    if (x mod k) = 0  then ancora := false;
                k := k+1
          end ;
            if  ancora  then   primo3 := true  else primo3 := false
   end ;

qui sotto iniziano le istruzioni del programma

begin
 writeln (1, "  e` primo");
 writeln (2, "  e` primo");
 for n := 3 to MAXNUM do   if primo3(n) then writeln (n, " e` primo")
end.
```

Nota.

E` importante capire che l'esecuzione del programma primi, inizia dalla prima istruzione del programma, cioè da writeln (1, " e` primo")

Quando poi si esegue l'istruzione for, allora (per n = 3, 4, ..., MAXNUM) viene chiamata la funzione primo3 sull'argomento n ; a questo punto si passa ad eseguire il corpo della funzione "sostituendo n al posto di x" (in effetti, questo è un punto delicato che vedremo meglio in seguito); questa esecuzione termina eseguendo una istruzione return che fornisce un valore al programma, valore che è usato, dal programma, per decidere se eseguire il ramo then, ...

Tuttavia, per capire qual'è l'effetto della chiamata primo3(n) non è necessario, e non è nemmeno consigliabile, cercare di simulare la sua esecuzione. E` molto più utile e sicuro un ragionamento ad alto livello, che segue proprio il modo in cui abbiamo introdotto la funzione:

per come è stata progettata, primo3 prende il suo argomento (che si suppone sia ≥ 3) e controlla se è primo o meno; se è primo produce il risultato "vero", altrimenti produce "falso";

allora, quando nel programma scriviamo "primo3(n)" questa espressione produrrà "vero" se n è primo, "falso" altrimenti; tutto qui (ma basta e avanza, per ora).

Vediamo infine come si scriverebbero le cose in C. Rispetto al Pascal, cambia un po' la forma, ma non la sostanza; qui seguiamo lo stile suggerito in [KR] e usiamo prototipi di funzione (vedere sotto).

Ricordiamo: 0 ↔ falso, ogni altro intero ↔ vero

Programma in C:

```
#include <stdio.h>
#define  MAXNUM  100

typedef  char  Bool   /* Bool e` una abbreviazione per booleani */

Bool primo3 (int)     /* questo è il prototipo della funzione
                  specifica dominio, int, e codominio, Bool   */
```

```
main()
     { int n;
       printf(" %d e` primo \n ", 1);
     printf(" %d e` primo \n ", 2);
     for(n=3; n<=MAXNUM; n++)
        {  if ( primo3 (n) )    printf(" %d e` primo \n ", n);
           else  printf(" %d non e` primo \n ", n);
        }
     }

Bool primo3 (int x)   /* questa è la vera def. della funzione */
         {    int k;
              for(k=2; k*k <= x ; k++) { if((x % k)  == 0) break; }
              if (k*k > x) return(1);  else return (0);
                        /* meglio :  return ( k*k > x )  */
         }
```

Chiariremo più avanti, nella sezione dedicata alle regole di visibilità delle dichiarazioni, il ruolo dei prototipi di funzione; qui ci limitiamo ad osservare che il prototipo permette di rispettare la regola "non si guarda avanti" e nello stesso tempo di strutturare il file che contiene il `main` e diverse procedure/funzioni in modo più leggibile.

Ci sarebbero ancora diverse cose da dire, però è utile prima avere un esempio di procedura.

16.2 . Parametri formali e parametri attuali

Nel contesto dei linguaggi di programmazione, per le funzioni, e ancor più per le procedure, si usa spesso la parola *parametro* invece della parola *argomento* (e non si usa mai la parola *variabile*, perché le variabili nei linguaggi di programmazione (imperativi) sono altra cosa). Inoltre, i nomi degli argomenti usati nella dichiarazione (x nel caso di primo3) si chiamano *parametri formali*, mentre gli argomenti al momento della chiamata (n nell'esempio dei primi) si chiamano *parametri attuali*.

I parametri formali sono sempre identificatori, mentre quelli attuali possono essere espressioni; in questo non c'è niente di nuovo: nessuno scrive: sia f la funzione t.c. f(x+3) = ...; ma, avendo definito f, si può scrivere f(a+b*3).

Come in matematica, i parametri formali, in quanto *nomi*, sono "irrilevanti"; per intenderci, la (1) e la (2) definiscono la stessa funzione:

(1) $f : \mathbf{N}^2 \rightarrow \mathbf{N}$ t.c. $f(x, y) = (x+2) * y$

(2) $f : \mathbf{N}^2 \rightarrow \mathbf{N}$ t.c. $f(n, p) = (n+2) * p$

Per evidenziare questo fatto si potrebbe scrivere:

(3) $f : \mathbf{N}^2 \rightarrow \mathbf{N}$ t.c. $f(\mathbf{[\,]}, \mathbf{\{\,\}}) = (\mathbf{[\,]} + 2) * \mathbf{\{\,\}}$

intendendo che **[ ]** e **{ }** sono "buchi" da riempire con valori al momento di usare la funzione.

16.3 . Un esempio di procedura

Ritorniamo all'ordinamento di una successione usando il metodo selection-sort.

Problema: in input un intero $n \geq 1$ e n numeri razionali $a_1, ..., a_n$

in output $b_1, ..., b_n$ tali che la successione $b_1, ..., b_n$ è una permutazione della successione $a_1, ..., a_n$ ordinata, in modo crescente ($b_1 \leq b_2 \leq ... \leq b_n$).

Idea dell'algoritmo:

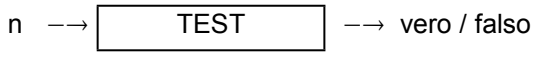
1. si memorizzano i numeri da ordinare in un array, chiamiamolo a, con indici da 1 a MAX, mettendoli nelle posizioni da 1 a n
2. si riordinano gli elementi dell'array da 1 a n col metodo detto selection-sort

3. si stampano gli elementi di a (che ora sono in ordine).

Il tutto è scritto in modo da evidenziare che ci sono tre fasi.

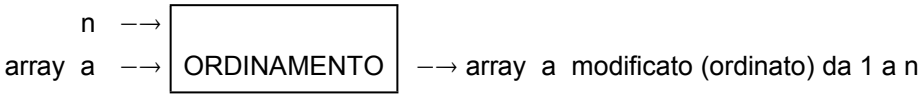
Quando poi abbiamo scritto il "programma" le tre fasi si potevano distinguere solo perché avevamo messo dei commenti; in molti casi può essere sufficiente, in altri no. Inoltre, e questo è ancora più importante: può capitare di dover ordinare un array, anche in altri contesti. Per questi motivi si preferisce isolare la parte di ordinamento anche a livello di codice, proprio come si è fatto per il test di primalità con la funzione primo3. (E` meno significativo, isolare la parte lettura dell'input e scrittura dell'output, perché è maggiormente legata al problema specifico, anche se negli esempi che facciamo ci sono molte similitudini.)

Nel caso dei primi, quello che ci serviva era: dato un numero n decidere se è primo o no; schematicamente:



che corrisponde precisamente allo schema di una funzione.

Se schematizziamo in modo analogo la fase di ordinamento dell'array, abbiamo:



Questo non è più uno schema funzionale (le funzioni per bene non modificano i loro argomenti), ma uno schema di *procedura*. In breve si può dire che:

- si usa una funzione per incapsulare tutto il lavoro necessario a "fornire un valore (e basta)", in particolare senza modificare gli argomenti;
- si usa una procedura per incapsulare il lavoro necessario a ... "eseguire un certo compito (ben preciso)"; questo compito può essere, appunto, riordinare un array, ma anche stamparlo o altro; in effetti una procedura può fare tutto quello che fa un programma.

La differenza tra "fornire un valore" e eseguire un certo compito" si riflette nel fatto che le chiamate di funzione compaiono all'interno di espressioni (come abbiamo visto), mentre quelle di procedura sono delle istruzioni (come vedremo).

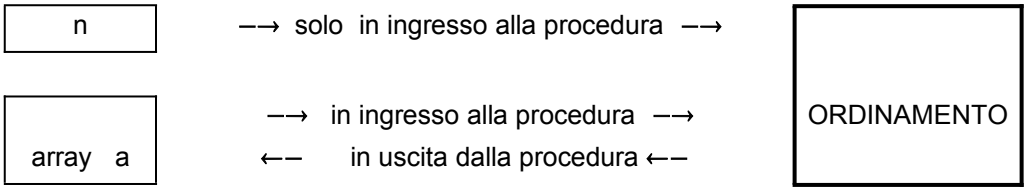
16.3.1 . Parametri IN, OUT, IN-OUT

Ritorniamo al discorso di modificare o meno gli argomenti, perché è importante.

Le funzioni (che seguono lo stile matematico) usano gli argomenti solo in ingresso, possiamo dire: solo in lettura (read only).

Le procedure li possono usare in più modi: solo in ingresso, solo in uscita (write only), sia in ingresso che in uscita. Per distinguere queste tre possibilità, per ogni argomento di una procedura, specificheremo se è IN (solo in ingresso), OUT (solo in uscita), IN-OUT (in ingresso ed in uscita).

Per capire meglio questo punto, un disegno più corretto per schematizzare la fase di ordinamento dell'array, è:



Come esempio di procedure con parametri (argomenti) OUT, cioè solo in uscita, abbiamo le procedure che realizzano le istruzioni di lettura (read, readln del Pascal, scanf ed altre in C). Questo si capisce ricordando che lo scopo di " leggi (sorgente, x) " è quello di assegnare ad x il valore prelevato da sorgente; il valore di x prima di eseguire leggi (sorgente, x) non interessa e verrà cancellato dal nuovo; quindi per la procedura che realizza leggi(sorgente, x), il parametro x è parametro OUT.

Non tutti i linguaggi distinguono le tre possibilità, come vedremo più avanti.

16.3.2 . Procedura selsort

Ritornando al nostro esempio, l'algoritmo per ordinare una successione di numeri, usando una procedura, diventa:

Algoritmo

costanti: MAX = 100 *dimensione massima degli array*

variabili: n : integer *quanti elementi bisogna leggere*

a : array [1 .. MAX] of real

k1 : integer *indice per scorrere su a*

istruzioni

leggi (n)

if (n > MAX) or (n < 1) then scrivi ("errore ....")

else { per k1 = 1, 2,..., n : leggi(a[k1]) *memorizziamo la successione nell'array*

selsort(a, n) ***riordiniamo a chiamando la procedura selsort***

per k1 = 1, 2,..., n : scrivi(a[k1]) *stampiamo il risultato*

}

Dove la procedura selsort è (in altri termini, la *dichiarazione* della procedura è):

```
Procedura selsort (     x : array [1 .. MAX] of real      parametro IN-OUT
                        dim: integer                      parametro IN        )
   la procedura ordina l'array x da x[1] a x[dim] quindi si suppone che sia: dim < MAX
   variabili:       k2, j, imin : integer
                    aux : real
   istruzioni:
         per k2 = 1, 2, ..., dim-1 : {
                    trova minimo in x da k2 a dim; chiamiamo imin l'indice di questa posizione:
                    imin ← k2
                    per j = k2+1, k2+2, ..., dim :      if x[ j ] < x[imin] then imin ← j

                    scambia x[k2] con x[imin]:
                    aux ← x[k2] ;   x[k2] ← x[imin] ;   x[imin] ← aux
         }   ora l'array x è ordinato da 1 a dim
```

Notare:

- j, imin, aux che prima erano nel programma ma servivano solo per la fase di ordinamento sono stati spostati dentro la procedura;
- nel programma visto precedentemente, senza procedure, usavamo l'indice k per parti che ora sono nelle procedura; per chiarezza qui abbiamo usato k1 in quello che resta nel programma e k2 per la procedura; vedremo in seguito (visibilità delle dichiarazioni) che si sarebbe potuto usare sempre k senza provocare pasticci.

L'effetto della chiamata **selsort(a, n)** all'interno del programma, o dell'algoritmo, chiamante si può spiegare in vari modi. Come per il caso dei numeri primi, scegliamo di ragionare "ad alto livello".

Per prima cosa: la chiamata ad una procedura è una istruzione; quindi si tratta di spiegare qual'è l'effetto di eseguire l'istruzione selsort(a,n). Ci sono essenzialmente 3 passi.

1. La procedura "legge" i parametri specificati come IN e IN-OUT (come ciò avvenga dal punto di vista implementativo non ci interessa ora); a questo punto è come se l'array a venisse copiato nell'array x ed il valore di n in dim.
2. Si esegue il corpo della procedura. In base a quello che abbiamo visto quando abbiamo discusso l'ordinamento per selezione, l'array viene riordinato da 1 a dim (=n).
3. La procedura "scrive" i parametri OUT (qui non ce ne sono) e IN-OUT; a questo punto è come se l'array x venisse ricopiato nell'array a. Quindi ora nel programma l'array a è riordinato. Dopo aver eseguito l'istruzione selsort(a,n)., il programma continua con l'istruzione successiva.

Per quanto riguarda la posizione della dichiarazione di procedura all'interno di un programma: tutto come per le funzioni. Nella Sezione 16.5 vedremo comunque lo schema della versione C.

16.4 . Ancora su IN, OUT e IN-OUT; parametri per valore e per riferimento

La distinzione tra IN, OUT, IN-OUT può sembrare un po' noiosa; inoltre non è presente, al completo, in quasi nessun linguaggio usato, ad eccezione di Ada e derivati. C'è almeno una buona ragione per insisterci. Quando si progetta una procedura è molto importante aver ben chiaro in testa il ruolo di ciascun parametro; la distinzione tra IN, OUT ed IN-OUT è un modo di chiarire certi aspetti. Visto al rovescio, quando si legge una procedura scritta da un altro, le specifiche IN,.... aiutano a capire cosa succede.

In Pascal si distingue tra parametri *per valore*, che corrispondono ai parametri IN, e parametri *per variabile* (o *per riferimento*, o *per indirizzo*), che corrispondono a quelli IN-OUT; quindi a livello di codice Pascal i parametri OUT vengono trattati come IN-OUT (se si vuole precisare che in realtà sono OUT, lo si fa usando dei commenti).

Spesso, invece di parametro per valore / riferimento, si dice "parametro *passato* per valore /riferimento" (e quindi di parla di *passaggio dei parametri*).

Il C ha solo parametri per valore, cioè IN; quelli OUT e IN-OUT si ottengono usando *puntatori* e indirizzi; in maniera implicita per gli array (che quindi vengono sempre trattati come IN-OUT), in maniera esplicita per gli altri casi.

Per prima cosa chiariamo il meccanismo per valore (identico in Pascal e C).

Quando abbiamo introdotto le procedure, abbiamo spiegato l'idea di parametro IN, dicendo che al momento della chiamata è come se la procedura leggesse il valore del parametro (attuale). Poiché in generale il parametro attuale può essere una espressione questo vuol dire "calcolare il valore dell'espressione e leggerlo". Un altro modo di vedere la cosa è illustrato dal disegno che segue.

La procedura P ha un unico parametro x per valore. Al momento della chiamata, con parametro attuale exp, si valuta il valore di exp e lo si "passa" alla procedura, copiando il valore un una cella predisposta per x.

In base a questo schema dovrebbe essere chiaro il comportamento di questo programmino:

```
#include <stdio.h>

void pp ( int x );

main () {    int  aa;
          aa = 3;
          pp(aa);
          printf ("%d" , aa) ;       /*  output : 3   NON  4  */
      }

void pp ( int x )    {  x++ }
```

La procedura pp cambia il valore contenuto nella "cella per x", ma questo non ha alcun effetto sul valore di aa nel programma.

Vediamo ora il passaggio per riferimento (Pascal, C++, Java, ...).

Per prima cosa, il parametro attuale non può essere una generica espressione, ma deve essere una variabile (anche con nome complesso come sss.nome[exp]), perché deve avere un indirizzo.

In genere si spiega il passaggio per riferimento dicendo che invece di passare alla procedura il valore del parametro attuale, supponiamo sia aa, si passa il suo indirizzo; quindi la procedura ha direttamente accesso alla cella per aa ed è in grado di modificarne il contenuto; quando l'esecuzione della procedura ha termine, il programma continua ed il valore di aa è quello modificato dalla procedura.

Per capire come funzionano veramente le cose, il modo più semplice è vedere come si fa in C. Infatti, il C non fornisce il passaggio per riferimento e quindi bisogna simularlo, facendo "a mano" quello che l'implementazione del Pascal fa automaticamente.

Vediamo un esempio (stupidissimo): vogliamo un programma che legge aa e stampa -aa (con aa intero). Per fare le cose in grande, definiamo una procedura che calcola l'opposto:

```
procedura opp (  x : intero    parametro IN;          y: intero      parametro OUT )
                    {     y ← - x  }
```

Programma in C:

```
#include <stdio.h>

void opp ( int x ,  int * py );

main () {    int  aa, bb;
          leggi( aa )       vedere oltre
          opp (aa, &bb);         /*  ora  bb = -aa  */
          printf ("%d  %d" , aa, bb) ;
       }

void opp ( int x ,  int * py )    {  *py = -x ;  }
```

Ignoriamo per il momento come si fa a leggere il valore per aa e concentriamoci sulla procedura opp.

Ha due parametri (formali): x, che è un int, e py, che è un puntatore_a_int; entrambi sono, a livello di linguaggio C, parametri per valore.

Il corpo della procedura dice:

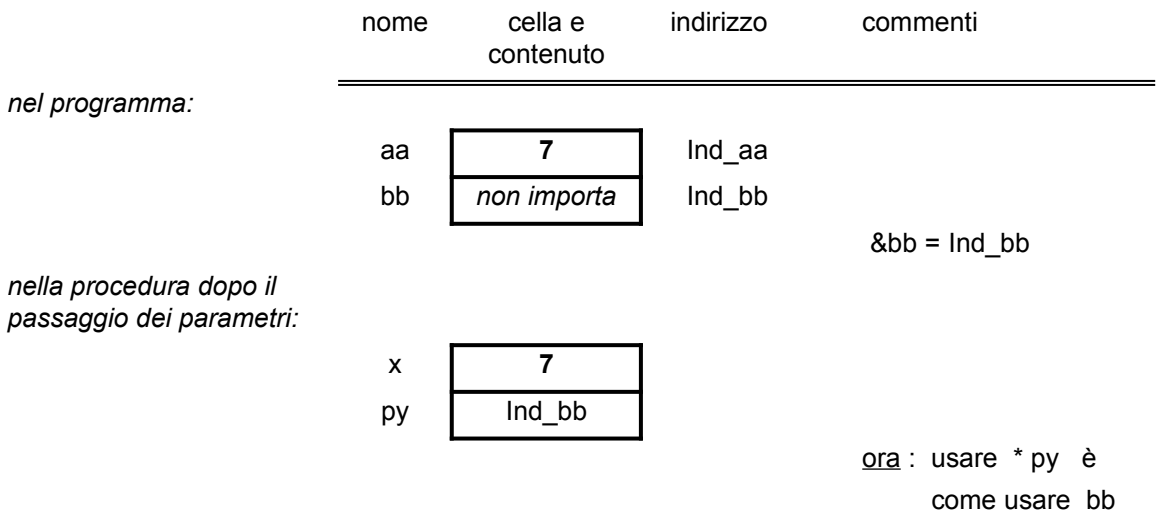
prendi il valore di x, cambia segno, vallo a mettere nella cella puntata da py, cioè nella cella il cui indirizzo è il valore di py

Nella chiamata, opp(aa, &bb), si passa alla procedura il valore di aa e il valore dell'espressione &bb, ma quest'ultimo è proprio l'indirizzo di bb. Quindi:

prendi il valore di x (dopo la chiamata è uguale al valore di aa), cambia segno,

vallo a mettere nella cella il cui indirizzo è il valore di py (dopo la chiamata è uguale a &bb), cioè nella cella per bb.

Schematicamente (supponendo di aver letto, nel main, il valore 7 per aa):



Un altro esempio che si fa sempre per capire l'uso di puntatori ed indirizzi è la procedura swap; vedere [KR] sezione 5.2, oppure [DM] sezione 9.1.

Vediamo ora la lettura di aa. In C, al posto di *leggi(aa)* si scrive: `scanf ("%d" , &aa);`

Per prima cosa, osserviamo che `scanf ("%d" , &aa)` è la chiamata ad una funzione di libreria che gestisce la lettura dal file "standard input"; il primo parametro è una stringa che descrive in modo un po' misterioso il formato dei dati; ma è il secondo che ci interessa. Dal punto di vista logico è un parametro solo OUT (il valore prima della chiamata non interessa, mentre interessa il valore dopo la chiamata); in C il meccanismo OUT si realizza allo stesso modo di quello IN-OUT, usando esplicitamente puntatori ed indirizzi; `&aa` è il parametro attuale che corrisponde ad uno formale di tipo puntatore; in questo è analogo a `&bb` nella chiamata `opp (aa, &bb)`.

16.5 . Array come parametri in C

Ritorniamo all'esempio di ordinamento per selezione. Lo schema della versione C è il seguente:

```
#include ......

#define MAX 100   /* la dimensione massima per l'array da ordinare */

void selsort( float x [ ], int dim );    /*  prototipo di  procedura    */

main( )
    {   int n  .............
        float a[MAX];       /*  l'array da ordinare                  */
        ..................... /*  lettura di  n  e degli  n  elementi .... */

        selsort (a,n);

        .....................  /*  stampa del risultato                */
    }

void selsort( float x[ ], int dim )    { ..................... }
```

Nell'algoritmo: il secondo parametro l'avevamo specificato come IN e qui è, correttamente, per valore; il primo parametro l'avevamo specificato come IN-OUT e qui sembrerebbe anch'esso per valore. Non è così: abbiamo visto che dopo una dichiarazione di array, come `float a[MAX];`

a è un puntatore al primo elemento dell'array; quindi nella chiamata `selsort(a,n);` viene passato il valore di `a` come puntatore, cioè l'indirizzo di `a[0]`.

Inoltre, nella dichiarazione della procedura non è necessario specificare le dimensioni dei parametri che sono array; quindi abbiamo scritto `float x[ ];`

Questo è molto comodo: la procedura `selsort` può ordinare array di lunghezza arbitraria.

Notare che quando si scrive, come sopra, `float x[ ]` l'unica cosa non specificata è quanti sono gli elementi, cioè qual'è il valore massimo per gli indici; per il resto: gli elementi sono di tipo `float` e gli indici sono interi, a partire da 0. A livello di pseudo-codice, può essere comodo usare indici di tipo diverso (ad esempio: 36, 37,..., 60 oppure 'a', 'b',..., 'z'); per mantenere questa libertà siamo però costretti a precisare quali indici usiamo. Ritorneremo su questo quando parleremo di array dinamici.

16.6 . Ancora sulla distinzione tra procedure e funzioni

La distinzione tra funzioni e procedure che abbiamo discusso sopra è (abbastanza) chiara a livello di metodologie di programmazione, ma in certi linguaggi, in particolare in C, è molto sfumata.

Nei manuali che parlano di Pascal c'è in genere la raccomandazione di programmare le funzioni in modo che siano "funzioni pure" (cioè che si comportino come in matematica, producendo un valore e basta), però a livello di linguaggio niente vieta di scrivere una funzione che fa cose folli.

In C la cosa è peggiorata dal fatto che le procedure sono di fatto "funzioni che non producono valori". Infatti l'intestazione di una funzione ha la forma :

<tipo del risultato> <nome> (<lista di argomenti >);

quella di una procedura ha la forma : **void** <nome> (<lista di argomenti >);

dove void è un tipo speciale che non ha valori.

Senza contare che in C anche il programma principale è una funzione, che ha un nome speciale: `main`.

Noi cercheremo di mantenere la distinzione, almeno a livello di algoritmo, come pure di mantenere, parlando di procedure, la distinzione tra IN, OUT e IN-OUT; per le funzioni useremo solo parametri IN, quindi è inutile precisarlo ogni volta.

16.7 . Tipo dei parametri e dei risultati

In matematica, qualunque insieme può essere dominio o codominio di una funzione, in altre parole non ci sono restrizioni sul tipo degli argomenti e dei risultati della funzioni. Nei linguaggi di programmazione, spesso le restrizioni ci sono.

Noi, negli algoritmi useremo lo stile della matematica, senza preoccuparci delle eventuali limitazioni imposte dai linguaggi concreti.

In C la situazione è la seguente.

- Il risultato di una funzione può essere dei seguenti tipi: aritmetici (char, int, float, long, ... signed, unsigned, ...), struct, union (cioè uno dei tanti tipi struct o union), void (allora si tratta di una procedura), puntatore a .... ;
  ma non può essere una funzione / procedura o un array (però può essere un puntatore a ....).
- Per gli argomenti non ci sono limitazioni: tutti i tipi (ammessi dal linguaggio) vanno bene. Come vedremo, si possono anche avere argomenti che sono funzioni / procedure.
- Tutti gli argomenti vengono passati per valore, ad eccezione di quelli che sono array o funzioni / procedure.

Le limitazioni sul tipo dei risultati, fanno si che, se non si vuole ricorrere ai puntatori, in alcuni casi una funzione venga simulata usando una procedura.

16.8 . Simulazione di funzioni usando procedure

E' sempre possibile simulare una funzione usando una procedura e il procedimento è standard, quasi meccanico. Dal punto di vista matematico, una funzione n-aria, cioè con n argomenti altro non è che una relazione (n+1)-ria con certe proprietà. La trasformazione si basa proprio su questo; vediamo lo schema, per brevità con n=2.

function f (x: integer , y: char): boolean dichiarazioni ..... istruzioni ..... return (exp)	procedure pf (x: integer, y :char -- IN res : boolean -- OUT) dichiarazioni ..... istruzioni ..... res ← exp; **exit** .....

La trasformazione è semplicissima: i parametri della funzione restano, come parametri IN; si aggiunge un nuovo parametro, con il tipo del risultato della funzione, che è parametro OUT; si rimpiazzano le istruzioni return con una coppia: assegnazione al parametro risultato ed istruzione che provoca l'uscita immediata dalla procedura ed il ritorno al programma chiamante; qui abbiamo usato **exit** ; a seconda del linguaggio bisogna sostituirla con istruzioni opportune; in C si scriverebbe semplicemente **return** (senza argomenti).

16.9 . Ancora un esempio: una funzione di conversione.

Abbiamo visto che gli array in C possono avere solo indici interi e che iniziano da 0. In alcuni casi questo non va bene. Vediamo come si può rimediare e contemporaneamente diamo un esempio di uso delle istruzioni case/switch.

Supponiamo di voler definire un tabella delle distanze tra Genova e le città d'Italia capoluoghi di provincia. Quindi vorremmo l'equivalente della tabella seguente, dove xxxx, yyyy,... sono degli interi che forniscono le distanze (calcolate in qualche modo):

Agrigento	xxxx
Alessandria	yyyy
.....	**.....**
Viterbo	zzzz

Supponiamo inoltre che le tabella venga usata solamente in questo modo: dato un nome di città si va a guardare la distanza corrispondente.

Un modo di realizzare questa tabella è quello di usare un array di record, dove i record sono coppie: (stringa, intero). In astratto, c'è un altro modo: un array di interi (le distanze) indiciato sui nomi di città. Graficamente:

indici:	*elementi:*
Agrigento	xxxx
Alessandria	yyyy
.....	.....
Viterbo	zzzz

Purtroppo, non si può fare in C (e nemmeno in Pascal). Una soluzione è quella di utilizzare una funzione di conversione (o anche di *hashing* bigettivo), che permetta di mappare la successione

delle città, ordinata, per esempio, in ordine alfabetico, sull'intervallo 0 .. NumCitta-1 (dove NumCitta è il numero di città nella tabella originaria).

Quindi, avendo definito un tipo Stringa, si definisce una funzione

function ind_citta (nome : Stringa) : integer

che restituisce un valore speciale, ad esempio -1, se nome non è tra quelli previsti e la sua posizione nella successione, altrimenti; ad esempio: ind_citta (Agrigento) deve produrre 0.

Per fare questo, il corpo della funzione può essere scritto usando una catena lunghissime di if-then-else, oppure usando una istruzione del tipo case/switch.

Vediamo prima una versione in pseudo-codice, poi la versione C, ricordando che nome è il parametro della funzione.

Versione in pseudo-codice:

casi su nome **:**

"Agrigento" → return(0)

"Alessandria" → return(1)

.....

"Viterbo" → return(NumCitta-1)

nessuno dei precedenti → return(-1)

fine

Notare che si tratta di una unica istruzione (composta).

Versione equivalente usando if-then-else che spiega il significato della precedente:

```
if nome = "Agrigento" then  return( 0 )
else    if  nome = "Alessandria" then  return( 1 )
        else .....................
                                    if  nome = "Viterbo" then  return( NumCitta-1)
                                    else return ( -1)
```

Versione in C (è sempre un'unica istruzione):

```
switch (nome)
{   case   "Agrigento" :   return( 0 );
    case   "Alessandria"   :   return( 1 );
    ..........................................
    case   "Viterbo"       :   return( NumCitta-1);
    default                :   return ( -1) ;
}
```

Attenzione: Qui, a destra del **:** c'è sempre una istruzione return; appena si esegue si esce dallo switch e dalla funzione. In generale le cose vanno diversamente: ad esempio, l'effetto di

```
switch (nome)
{   case   "Agrigento" :   scrivi( 0 );
    case   "Alessandria"   :   scrivi( 1 );
    ..........................................
    case   "Viterbo"       :   scrivi( NumCitta-1);
    default                :   scrivi ( -1) ;
}
```

sarebbe quello eseguire tutte le istruzioni di destra, a partire dal caso che ha avuto successo; quindi, con nome = “Alessandria”, si avrebbe la stampa di 1, 2, ..., NumCitta-1, -1.

Per ottenere l’effetto stile if-then-else, bisogna usare delle istruzioni break:

```
switch (nome)
{   case   “Agrigento” :    scrivi( 0 );   break ;
    case    “Alessandria”   :   scrivi( 1 );    break ;
    .....
    case   “Viterbo”        :   scrivi( NumCitta-1);   break ;
    default                 :   scrivi ( -1) ;
}
```

16.10 . Funzioni e procedure di libreria

Non è sempre necessario scrivere le funzioni e le procedure di cui abbiamo bisogno; esistono delle *librerie* di funzioni e procedure predefinite. Queste comprendono:

- funzioni matematiche standard (es: radice quadrata, logaritmo, esponenziale, funzioni trigonometriche);
- funzioni (e procedure) per manipolare stringhe; come minimo: uguaglianza, maggiore e minore (ordine lessicografico);
- funzioni e procedure per gestire file;
- *utilities*, ad esempio procedure per l’ordinamento di array.

Le varie implementazioni del C offrono almeno la Standard library. Questa è divisa in “sezioni” corrispondenti a diversi file “header”. Le misteriose righe della forma `#include .....` che si trovano all’inizio dei programmi C servono appunto a segnalare all’implementazione che nel programma si useranno una o più sezioni (o meglio: funzioni appartenenti a quelle sezioni). Ulteriori dettagli nelle lezioni sul C e in [KR] o [DM].

16.11 . Esercizi

Esercizio 1.

Somma e prodotto di vettori e matrici si possono formulare sotto forma di procedure (non come funzioni, a meno di usare puntatori per il risultato). Provare.

Esercizio 2

Il crivello di Eratostene fornisce un modo di trovare i numeri primi da 1 ad un certo valore massimo, sia MAX. Ci sono vari modi di presentarlo; uno è il seguente.

- Si parte con la successione di tutti gli interi da 1 a MAX : 1, 2, 3, ..., MAX
- Sappiamo che 1 e 2 sono primi e li lasciamo stare.
- Cancelliamo tutti i multipli di 2 (cioè: 4, 6, 8, ...).
- Andiamo a prendere il primo numero rimasto maggiore di 2; questo numero (che è 3) è primo (non perché sappiamo che 3 è primo, ma per la ragione che vedremo); cancelliamo tutti i suoi multipli (cioè: 6, 9, 12,....). Facendo così ci troviamo a cancellare numeri, come 6 o 12, già cancellati; pazienza.
- Passo generico.
  Sia p il numero primo i cui multipli sono stati cancellati al passo precedente.
  Consideriamo il primo numero maggiore di p rimasto nella successione; chiamiamolo q.
  Chiaramente, q non è multiplo di nessuno dei numeri (primi) che lo precedono (altrimenti sarebbe stato cancellato), dunque anche q è primo. Cancelliamo tutti i suoi multipli (2q, 3q, ...)

- Quando si è trovato un q abbastanza grande (quanto ??), rispetto a MAX, ci si può fermare: tutti i non-primi sono stati cancellati e restano solo i primi.

Si può realizzare il crivello usando un array criv di booleani con indici da 1 a MAX: criv[k] = vero vuol dire che k non è stato (ancora) cancellato.

Quindi, **l'esercizio è**: scrivere una procedura Crivello che realizza il tutto.

17. Regole di visibilità delle dichiarazioni

Qui vogliamo chiarire il "raggio di azione" (*scope*, in inglese) delle dichiarazioni; si dice anche: la visibilità delle dichiarazioni.

Prima, però aggiungiamo qualcosa a quanto visto sulle dichiarazioni di variabili.

17.1 . Dichiarazioni con inizializzazione

In C (ma non in Pascal), è possibile fornire un *valore iniziale alle variabili* al momento della dichiarazione.

Esempi: `int n = 3;` `float y = 7.7;` `char c = 'm'`

`int v [5] = { 10, 11, 12, 13, 14 }`

L'ultima è equivalente a: `int v[5];` *dichiarazione*

`v[0] = 10; v[1] = 11; .....` *inizializzazione*

17.2 . Il gioco dei rettangoli

Concentriamoci per ora sulle dichiarazioni di variabile (con inizializzazioni, perché permettono di capire meglio cosa succede). Quello che vediamo è abbastanza generale, in particolare vale per C e Pascal (salvo l'uso delle inizializzazioni); usiamo la sintassi del C perché è più concisa.

Consideriamo il disegno seguente:

R. 1

```
int x = 3 ;
int y = 5 ;
```

qui: x è 3 y è 5

R. 2

```
float  x = 3.14 ;
int z = 9 ;
```

qui: x è 3.14 y è 5 z è 9

R.3

```
int y = 10 ;
int w = 30 ;
```

qui: x è 3.14 y è 10 z è 9 w è 30

qui: x è 3.14 y è 5 z è 9 w è sconosciuto

qui: x è 3 y è 5 z, w sono sconosciuti

R.4

```
int w = 27;
int x = 88;
```

qui: x è 88 y è 5 w è 27z è sconosciuto

qui: x è 3 y è 5 z, w sono sconosciuti

Livelli:

R.1 è a livello 0

R.2, R.4 sono a livello 1

R3. è a livello 2

(i livelli sono utili per il C)

Vediamo le regole del gioco:

1. I rettangoli: o sono completamente disgiunti, come R.2 ed R.4, oppure sono uno dentro l'altro, come R.1 e R.2. Questi rettangoli rappresentano: tutto il programma e blocchi o procedure e funzioni contenuti in esso, come vedremo meglio in seguito.
2. Una dichiarazione, con identificatore <id>, è *visibile* (oppure: è *valida*):
   - nel rettangolo dove compare (ad esempio: int z = 9 compare in R.2) e
   - in tutti i sottorettangoli (ad es. int z = 9 vale anche in R.3)
     a meno che non venga "coperta" da una altra dichiarazione di <id>; ad esempio:
     int x = 3 compare in R.1; in R.2 è coperta dalla nuova dichiarazione di x (float x = 3.14);
     in altre parole la x di R.1 e quella di R.2 sono due variabili indipendenti.
3. La visibilità è a senso unico, dall'interno verso l'esterno; ad esempio: R.3 vede quello che è stato dichiarato in R.2 ed anche in R.1 (a meno che non sia coperto da qualche dichiarazione in R.2), ma né R.2, né R.1 vedono quello che è stato dichiarato in R.3.

Dopo alcuni esempi, complicheremo il gioco e introdurremo un'altra regola: visibilità solo verso l'alto.

Nella terminologia comune:

un identificatore è *locale* al rettangolo dove è dichiarato, *globale* ai sottorettangoli dove è visibile; ad esempio; la z dichiarata da int z = 9 è locale a R.2 ed è globale a R.3.

In C si usa una terminologia diversa (variabili *automatiche* ed *esterne*); vedere [KR] e sotto.

Come si definiscono i rettangoli

In Pascal: il rettangolo più esterno corrisponde al programma; quelli interni alle procedure/funzioni.

In C, la situazione è un po' più complicata; per spiegarlo ritorniamo al disegno dei rettangoli; per brevità, seguiamo lo stile del C e parliamo solo di funzioni (le procedure sono funzioni a valori in void); allora (nell'ipotesi di avere un unico file sorgente):

- il rettangolo più esterno (livello 0) corrisponde al file; è terra di nessuno (o di tutti); lí si mettono gli `#include`, i `#define` (e quindi le "dichiarazioni di costante"), le dichiarazioni di variabile, le dichiarazioni di tipo, cioè i `typedef`, ed i prototipi di funzione che devono essere visibili a tutti i rettangoli (che stanno sotto queste dichiarazioni); nel gergo C, le variabili dichiarate qui vengono dette *esterne*; questo rettangolo non può contenere istruzioni;
- i rettangoli di livello 1 corrispondono alle funzioni, tra cui il `main`; in C le funzioni non possono essere annidate, cioè non è possibile definire una funzione all'interno di un'altra (come si può fare in Pascal); nel gergo C, le variabili dichiarate qui vengono dette *automatiche*;
- i rettangoli di livello ≥2 corrispondono ai blocchi con dichiarazioni, che possono essere annidati; nel gergo C, anche le variabili dichiarate qui vengono dette *automatiche*.

Vediamo un esempio.

```
#include <stdio.h>

int y = -1;

main()
 {
      int x = 2;
      printf ("%d   %d \n\n", x, y);    /*  output : 2  -1  */

              /* quello che segue e` un blocco con dichiarazioni */
   {  int y = 7;
      printf ("%d   %d \n\n", x, y);  /*  output : 2  7  */
   }

   printf ("%d   %d \n\n", x, y);   /*  output : 2  -1  */

 }
```

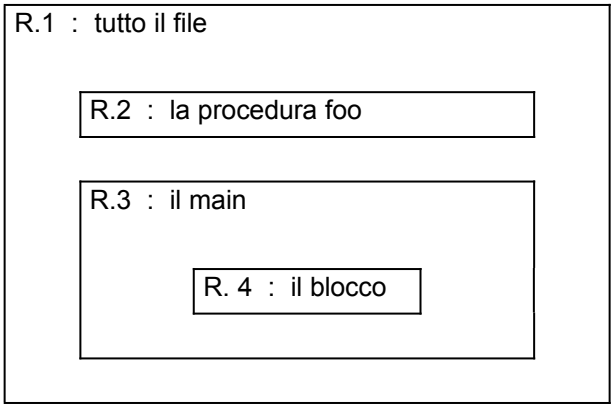
Qui ci sono 3 rettangoli: R.1 per tutto il file, R.2 per il main e R.3 per il blocco; R.1 contiene R.2 che contiene R.3.

Altri esempi, più complessi, che richiedono ulteriori regole (che vedremo subito dopo):

Esempio 1

```
#include <stdio.h>

int x = 1;
int y = -1;

void foo(void)
 {
    int y = -2;
    printf ("%d  %d \n\n", x, y);
 }

main()
 {
      int x = 2;
      printf ("%d   %d \n\n", x, y);    /*  output : 2  -1  */

   {  /* questo e` un blocco con dichiarazioni */
      int y = 7;
      printf ("%d   %d \n\n", x, y);  /*  output : 2  7  */
   }

   printf ("%d   %d \n\n", x, y);   /*  output : 2  -1  */

      foo();                        /*  output : 1  -2  */
 }
```

Qui abbiamo:



Esempio 2

```
#include <stdio.h>

int x = 1;
int y = -1;

void foo (void);   /* prototipo di funzione */
void fuu (int);    /* prototipo di funzione */
              /* se  li togliessimo main non vedrebbe ne' foo ne' fuu visto
                 che la loro dichiarazione compare dopo il main */

main()
 {
     int x = 2;
     printf ("%d   %d \n\n", x, y);    /*  output : 2  -1  */

   {  /* questo e` un blocco con dichiarazioni */
     int y = 7;
     printf ("%d   %d \n\n", x, y);  /*  output : 2  7  */
   }

   printf ("%d   %d \n\n", x, y);  /*  output : 2  -1  */

     foo();                    /* output : 1  -2  */

     fuu(x);                   /* output : 2  1  -1   17  */
 }


void foo(void)      /* questa e` la dichiarazione completa di foo */
 {
    int y = -2;
    printf ("%d  %d \n\n", x, y);
 }


int z = 17;    /* sarebbe vietato qui ridefinire  la  x  o la  y
            * solo  fuu vede questa dichiarazione di z
            */


void fuu(int k)   /* questa e` la dichiarazione completa di fuu */
 {
    printf ("%d  %d  %d  %d \n\n", k, x, y, z);
 }
```

Il gioco dei rettangoli e le regole viste finora non sono sufficienti a spiegare completamente gli ultimi due esempi; dobbiamo complicare il gioco.

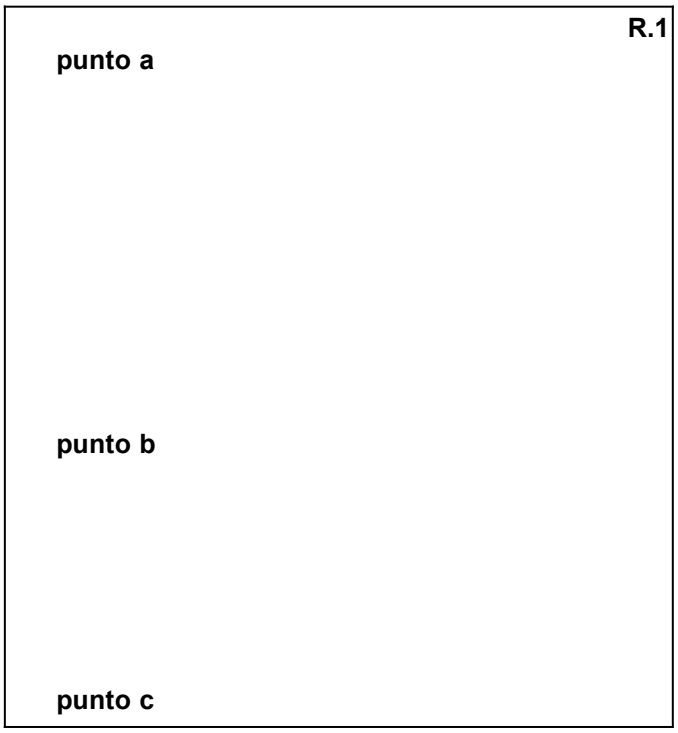
Per prima cosa, non possiamo limitarci a considerare solo le dichiarazioni di variabile; ci sono quelle di costante, di tipo, di procedura,...

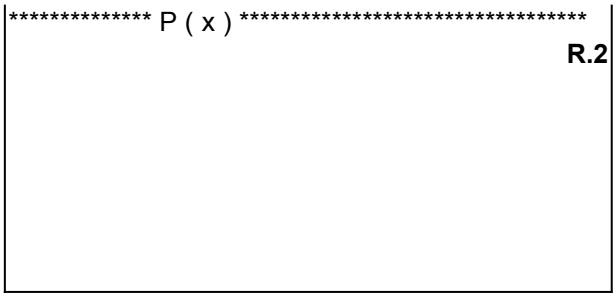
Le dichiarazioni di tipo seguono le regole viste per le variabili.

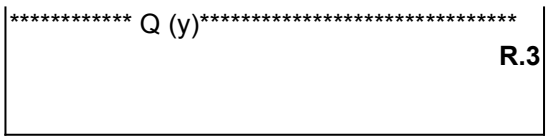
Le dichiarazioni di costante in Pascal seguono le regole viste per le variabili; quelle fatte in C attraverso i #define non seguono le regole viste, ma una regola molto semplice che vedremo più avanti.

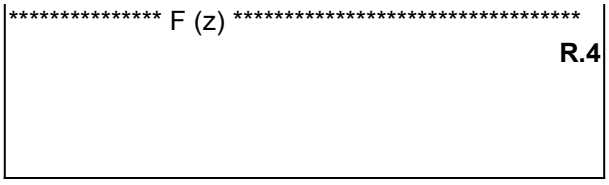
Il problema sono le dichiarazioni di procedura e funzione, perché corrispondono ad avere un nome e dei parametri per alcuni rettangoli.

Gioco modificato: i rettangoli possono avere un nome e dei parametri.
Vediamo un esempio semplice:









In questa situazione:

- i nomi dei rettangoli (P, Q, F) è come se fossero dichiarati nel rettangolo che li contiene (quindi: P ed F dichiarati in R.1 e Q, in R.2; quindi Q è visibile dentro R.2 ed è invisibile in R.1 ed in R.4);
  i parametri è come se fossero dichiarati nel rettangolo stesso (quindi: x dichiarato in R.2, y in R.3, z in R.4).

E' una regola più che sensata, pensando alle procedure/funzioni: il nome di una procedura o funzione deve essere visibile per poterla "chiamare". I parametri formali, invece hanno solo il ruolo di "contenitori".

Notare che in C non è possibile creare R.3, perché non si può dichiarare una funzione all'interno di un'altra.

- Ulteriore regola (per C e Pascal) : la visibilità è verso l'alto; quindi, riferendosi al solo rettangolo R.1: nel **punto a**, non si vede né P, né F (Q non si può vedere perché è dentro R.2); nel **punto b** si vede P, ma non F; nel **punto c** si vedono P ed F.

Per "rimediare" al vincolo che la visibilità è verso l'alto (ma non solo per questo), in C si possono usare **i prototipi di funzione** (e quindi anche di procedura); i prototipi sono presenti anche in Pascal (con parola chiave **forward**), ma con un uso più limitato.

Il prototipo dichiara semplicemente: nome della funzione, tipo degli argomenti (anche il nome dei parametri formali se si vuole, magari per poter inserire dei commenti), tipo del risultato. Dal punto di vista della visibilità, tuttavia, equivale ad una dichiarazione completa.

Quindi per la visibilità di g da parte del main, sono equivalenti i due programmi seguenti

```
#include ....

int  g (float x)
   {  ..... corpo di g }

main () {  int a;
         a = g(3.14);
         .............
       }
```

```
#include ....

int  g (float);

main () {  int a;
         a = g(3.14);
         .............
       }

int  g (float x)
   {  ..... corpo di g }
```

I prototipi, se ben commentati favoriscono la leggibilità del programma (hanno poi un ruolo importante nello sviluppo di programmi complessi suddivisi in molti file, cosa che non vedremo). Al limite, si dovrebbe poter capire cosa fa il main leggendo solo: main, dichiarazioni e prototipi che lo precedono; quindi senza guardare il codice delle altre funzioni.

La visibilità dei #define

La visibilità di un #define, e quindi il suo raggio di azione, inizia dalla riga che lo contiene e continua fino alla fine del file, scavalcando ogni barriera ed ignorando tutte le regole viste (tranne quella della visibilità verso l'alto). Quindi è anche vietato ridefinire l'identificatore. Il fatto è che #define non introduce una dichiarazione ma una *direttiva* (al pre-processore).

Ad esempio, #define UNO 1 è una direttiva che dice:

da questo punto in poi, cambia tutte le occorrenze di UNO (eccetto quelle che compaiono in stringhe o all'interno di identificatori) in 1

Vediamo un esempio:

```
#include <stdio.h>

void foo (void);
void fuu(void);

main()
 {
   #define UNO 1

   foo();          /* output  1    */

   fuu();          /* output  2    */
 }

/*   punto A : a livello di file  */

void foo(void)  {  printf ("%d  \n", UNO); }

/*   punto B : a livello di file  */

void fuu(void)     {   int x = UNO + 1;
                   printf ("%d  \n",x);
 }
```

Il nome UNO, benché definito dentro il main, è visibile dentro le altre funzioni, come pure nel punto A e nel punto B; è vietato ridefinire UNO (con un #define, o con una dichiarazione), sia nelle funzioni che in A o B.

17.3 . Variabili locali (automatiche)

Consideriamo il seguente programma C

```
#include <stdio.h>

void prrr (void);   /* prototipo */
              /* esempio di come NON si deve commentare un prototipo! */

main()
  {
    for (k = 1; k < 4; k++)
       {
         int n=1;
         printf ("%d  \n", n);  /*  output : 1  */
         n++;
         printf ("%d  \n", n);  /*  output : 2  */
         prrr ();
        }
  }

 void prrr(void)
  {
      int m = 5;
      printf ("%d \n", m);  /*  output : 5  */
      m++ ;
    printf ("%d \n", m);  /*  output : 6  */
   }
```

L'output prodotto dall'esecuzione del main è:

```
1       /*  1a stampa di  n  da main        per k=1  */
2       /*  2a stampa di  n  da main            " "   */
5       /*  1a stampa di  m  da prrr        " "   */
6       /*  2a stampa di  m  da prrr            " "   */
1       /*  1a stampa di  n  da main        per k=2  */
2       /*  2a stampa di  n  da main            " "   */
5       /*  1a stampa di  m  da prrr        " "   */
6       /*  2a stampa di  m  da prrr            " "   */

eccetera
```

Quello che succede è che ogni volta che si esegue il blocco corpo del for si "esegue" la dichiarazione int n =1 e quindi il valore precedente per n (cioè 2) viene perso. Del tutto analogo quello che succede per m, ad ogni chiamata di prrr. Quindi: ogni volta che si entra nel blocco corpo del for (ogni volta che si entra nel corpo di prrr, per effetto della chiamata) si crea una nuova variabile n (m in prrr) che cessa di esistere appena si esce dal corpo del for (da quello della procedura). In altre parole eseguendo il for si creano e distruggono 3 n e 3 m.

17.4 . Variabili globali "contro" parametri

Alcuni si sentono un po' a disagio con i parametri (tutti quei discorsi: IN, OUT,....) e preferiscono usare le variabili globali. Le regole viste, però limitano questa scappatoia; vediamolo nel caso del C. In effetti, questo si può vedere già negli esempi 1 e 2 visti prima; però essendo esempi "artificiali" la cosa non è evidente.

Supponiamo di avere un main che chiama due procedure f1 ed f2; supponiamo inoltre che tutte e due le procedure debbano ad un certo momento controllare se una loro variabile locale, di tipo intero, è un numero primo o meno. Volendo usare una funzione, chiamiamola primo, per fare il test, si potrebbe pensare di organizzare le cose in questo modo:

```
#include ...........

int quello;   /* qui serve solo a non far arrabbiare il compilatore
              nella spiegazione, per non far confusione, lo indicheremo con
              quello_0*/
.................   /* altre dichiarazioni */

void f1 (void);    /* prototipo  */
void f2 (void);    /* prototipo */
int primo (void)   /* al solito 1 = vero, 0 = falso */

main()
  {  ...................
     f1();
     ...................
     f2();
     ...................
  }

void f1(void)
  {  int quello;    /* è la variabile che vogliamo controllare
                    sotto la indicheremo con quello_1       */
     .......................
     if  ( primo() ) ........
     ..........................
  }

void f2(void)
  {  int quello;    /* è la variabile che vogliamo controllare
                    il trucco è usare sempre lo stesso nome
                    sotto lo chiameremo quello_2              */
     .......................
     if  ( primo() ) ........
     ..........................
  }

int primo ()
  {
    /* qui si fanno tutti i conti; il trucco è  usare ancora il nome
       "quello" e controllare, appunto, se quello è un primo         */
  }
```

Dovrebbe essere chiaro in cosa consiste il trucco: quando chiamo primo() all'interno di f1, quello di primo (cioè il nome quello usato nella funzione primo) si dovrebbe riferire a quello_1, mentre quando chiamo primo() all'interno di f2, si dovrebbe riferire a quello_2.

Sciaguratamente (?) non funziona: in base alle regole, quello di primo si riferisce sempre a quello_0.

Questo si capisce anche ragionando in questo modo:

quando in f1 si esegue la chiamata primo(), immaginiamo di andare ad eseguire il corpo della funzione primo, portandoci dietro tutte le informazioni relative ai parametri attuali (cioè niente); una volta che siamo "dentro" primo, a chi si riferisce quello ? ovviamente a quello_0.

In effetti, un modo di fare funzionare le cose c'è:

```
#include ...........

int quello_0;
.................   /* altre dichiarazioni */

void f1 (void);    /* prototipo  */
void f2 (void);    /* prototipo */
int primo (void)   /* al solito 1 = vero, 0 = falso */

main()
  {  ....................
     f1();
     ....................
     f2();
     ....................
  }

void f1(void)
  {  int quello;
     ........................
     quello_0 = quello
     if  ( primo() ) ........
     ........................
  }

void f2(void)
  {  int quello;
     ........................
     quello_0 = quello
     if  ( primo() ) ........
     ........................
  }

int primo ()
  {     /*   qui si usa quello_0 */    }
```

Funziona, ma è un po' contorto; molto meglio usare i parametri !

17.5 . A proposito della struttura dei programmi C

A questo punto, possiamo riconsiderare la struttura di un programma C e cercare di capirla meglio. Riprendiamo l'esempio 2, visto prima, ma sintetizzandolo.

```
#include <stdio.h>

int x = 1;    int y = -1;
void foo (void);
void fuu (int);

main()
  {   /* dichiarazioni e istruzioni (incluse le chiamate a  foo  e fuu ) */  }

void foo(void)
  {   /* dichiarazioni e istruzioni  */  }

int z = 17;

void fuu(int k)
  {   /* dichiarazioni e istruzioni  */  }
```

Un modo di far quadrare questo programma con lo schema visto all'inizio di queste note:

programma = intestazione + parte dichiarazioni + parte istruzioni

è quello di riscriverlo (a questo punto, però, non è più C); le aggiunte sono in **grassetto**:

intestazione

```
#include <stdio.h>
```

parte dichiarazioni

```
int x = 1;    int y = -1;
void foo (void);
void fuu (int);

int main(void)
  {   /* dichiarazioni e istruzioni (incluse chiamate a  foo  e fuu ) */  }

void foo(void)
  {   /* dichiarazioni e istruzioni  */  }

int z = 17;

void fuu(int k)
{   /* dichiarazioni e istruzioni  */  }
```

parte istruzioni

```
main() ;    /* una sola istruzione: la chiamata alla funzione main */
```

Note. Il main è una funzione come le altre; ha due sole caratteristiche speciali: il nome ed il fatto che l'esecuzione di tutto il file comincia chiamando il main.

Per il tipo di programmi che scriviamo, non si capisce perché il main sia una funzione a valori interi (in certe applicazioni invece è utile); quindi sarebbe più sensato dichiarare il main come **void** main(**void**) Tuttavia abbiamo preferito adeguarci alla tradizione ed a [KR].

18. Variabili statiche e dinamiche.

In C ed in Pascal, le variabili possono nascere in due modi.

1. Tramite una dichiarazione, come visto finora, ed allora vengono anche dette *variabili statiche*. (Attenzione: nel gergo C la specifica **static** ha un altro significato !!).

Al momento della dichiarazione si associa (almeno) un nome alla variabile e questo legame tra nome e variabile rimane stabile (all'interno del raggio di azione della dichiarazione).

2. Tramite un puntatore e l'uso della procedura **new** in Pascal, o delle funzioni **calloc** e **malloc** in C. Queste variabili sono dette *dinamiche*, per i motivi che vediamo ora.

Poiché le cose sono un po' più semplici in Pascal, iniziamo con il Pascal.

Se pun è un puntatore_a_T, eseguendo la chiamata new(pun) viene generata una variabile di tipo T e il valore di pun posto uguale all'indirizzo di questa variabile.

Schematicamente, con T = char:

			cella per pun	*celle generate con new*	*indirizzi di queste celle*
dichiarazione:		**var** pun : **^** char			
effetto:			**????**		
istruzioni:	(1)	new(pun)			
effetto:			**ind_1**	**????**	ind_1
	(2)	pun^ := 'z'			
effetto:			ind_1	**'z'**	ind_1
	(3)	new (pun)			
effetto:			**ind_2**	**????**	ind_2

L'effetto dell'istruzione (1) è, come detto, quello di creare una nuova cella, buona per un carattere, ed inoltre, se ind_1 è il suo indirizzo, di fare l'assegnazione pun ← ind_1.

La nuova cella ha un valore "casuale", indicato con ????.

A questo punto, il nome della cella all'indirizzo ind_1 è pun^ (*pun in C); quindi l'istruzione (2) mette 'z' nella cella.

Una nuova chiamata new(pun), sconvolge tutto, perché viene creata una nuova cella, ad un altro indirizzo, e si cambia il valore di pun. A questo punto la cella all'indirizzo ind_1 non è più raggiungibile.

Tutto questo non sembra molto interessante; ha invece applicazioni per la costruzione di strutture dinamiche: array (in C - vedere sotto), liste, alberi,... (in C e Pascal - vedere altro fascicolo dispense).

Notare infine che non sono le variabili puntatore ad essere dinamiche, ma quelle "puntate", di tipo carattere, nel nostro esempio.

Per capire meglio quanto segue, diamo un'idea di cosa succede eseguendo una generica istruzione new(p), senza entrare nei dettagli tecnici, anche perchè variano a seconda del linguaggio e dell'implementazione.

L'istruzione new(p) equivale alla richiesta di un blocco di k byte contigui (pari alla dimensione della "cella" che serve); questa richiesta è indirizzata ad un "sistema", che può essere il "sistema di supporto a run time del linguaggio" o il sistema operativo. Se il sistema trova il blocco: "segna" ciascun byte come "occupato" e, come risposta alla richiesta, fornisce l'indirizzo del primo byte; altrimenti segnala in qualche modo il fallimento, ad esempio, restituendo un valore speciale:

l'indirizzo nullo. A questo punto l'esecuzione del programma riprende (eventualmente gestendo la situazione di fallimento).

Tornando all'esempio di sopra, abbiamo visto che quando si esegue l'istruzione (3) il valore del puntatore pun diventa ind_2 e la cella di indirizzo ind_1 non è più raggiungibile. Però la cella viene considerata dal sistema ancora come "occupata". (Non è stupidità; in generale ci possono essere tanti puntatori che puntano alla stessa cella; anche dopo aver fatto new di uno, la cella è raggiungibile usando gli altri.) Per fare un altro esempio, dopo l'istruzione

```
per k = 1, 2, ..., 1000 : new(pun)
```

nello spazio di lavoro del programma ci sono (oltre al resto) 1000 celle generate da new(pun), di cui 999 inutilizzabili da parte del programma, ma anche da parte del sistema che continua a considerarle occupate.

Esiste l'antidoto: la procedura **dispose**. Nell'esempio di prima, si potrebbe scrivere:

```
new(pun)
per k = 2, 3, ..., 1000 : {   dispose (pun)      libera la cella attualmente puntata
                                 new(pun)          ne chiede una nuova
                              }
```

È, purtroppo, un esempio artificiale; ma per il momento ci dobbiamo accontentare.

L'effetto di dispose(....) è uno dei punti delicati del Pascal. Lo standard prescrive certe cose, ma le implementazioni spesso fanno altro.

Ritorniamo al primo esempio, e inseriamo una istruzione dispose(pun) tra la (2) e la (3).

Dal punto di vista del programmatore, l'effetto è:

- la cella di indirizzo ind_1 viene "liberata", cioè si segnala al sistema che può riutilizzarla; quindi il programma non può più utilizzarne il contenuto;
- il valore di pun è indefinito.

In realtà, con alcune implementazioni le cose vanno in maniera un po' diversa; però se si vogliono evitare pasticci, bisogna comportarsi come se fosse sempre così.

D'altro canto, per capire alcuni comportamenti anomali di programmi in cui, per errore, si cerca di utilizzare il puntatore per accedere alla cella liberata, conviene sapere che in alcune implementazioni l'effetto di dispose(pun) è soltanto quello di segnalare al sistema che la cella non interessa più e può essere riutilizzata; in particolare, il valore di pun rimane inalterato, come pure inalterato è il contenuto della cella. Per chiarire questo punto, vediamo un esempio:

```
var p, q : puntatori_a_interi
new(p)                          ora il valore di p è ind_1
p^ ← 3                          la cella all'indirizzo ind_1 ora contiene 3
dispose(p)
new(q)                          ora il valore di q è ind_2
q^ ← 4                          la cella all'indirizzo ind_2 ora contiene 4
scrivi(p^)
```

I risultati possibili dell'ultima istruzione (scrivi(p^)) sono:

- messaggio di errore o altra forma di protesta (l'implementazione vigila);
- output 3 : l'implementazione fa solo il minimo, cioè segnala che la cella è riutilizzabile, ma il valore di p non viene toccato;
- output 4 : come caso precedente, ma il sistema nel rispondere alla richiesta generata da new(q) è andato a prendere proprio la cella appena liberata (quindi ind_2 = ind_1); d'altro canto, il valore di p non è stato toccato, quindi .....

Ancora un esempio, per vedere il tipo di problemi cui si può andare incontro se non si fa attenzione.

```
var  p, q : puntatori_a_interi
new(p)                          ora il valore di   p   è  ind_1
p^ ← 3
q ← p                           ora anche  q  vale  ind_1
dispose(p)
new(p)                          ora il valore di  p  è  ind_2
p^ ← 4
scrivi(q^)                      3 possibili risultati, come prima
```

Conclusione: prima di usare un dispose(p), controllare bene la situazione.

Variabili dinamiche in C

Diamo solo l'essenziale, concentrandoci sulle differenze rispetto al Pascal; per dettagli ed approfondimenti, vedere [KR] o [DM].

Per ottenere nuove celle ci sono due funzioni di libreria: **malloc** e **calloc**; per liberarle c'è la procedura **free**. Il funzionamento è analogo a quello che abbiamo visto in Pascal, ma si possono chiedere blocchi contigui di celle. Vediamolo con un esempio.

```
#include <stdlib.h>     /* malloc, calloc e free sono nella libreria  stdlib */
.....................
float  * p, * q  ;      /* p, q sono puntatori a float  */
.....................

p = calloc (100 , sizeof(float) );

       /*  ora  p  punta  ad un array di 100 celle di tipo float;
           su questo array si lavora come su array normali, ved. nota [1],
           per dettagli su  calloc e sizeof vedere nota [2]       */

if (p = NULL)  {  /* non c'è abbastanza memoria libera; a seconda del programma:
                continuiamo in qualche modo, o  abbandoniamo      */
            }

else  {    /* ora usiamo le celle che ci siamo procurate; ad esempio: */

        (for k=0 ; k<100; k++)   *(p+k) = k*3.14 ;   /* vedi nota [2] */
        ....................................................

        /* ora abbiamo finito di lavorare con i multipli di 3.14  e liberiamo
           la memoria, liberando tutte e 100 le celle: */

        free(p);    /* vedi nota [3]   */

        .....................
}
```

Nota [1] Sugli array generati dinamicamente si può andare su e giù usando l'aritmetica dei puntatori, ma anche la notazione solita degli array; quindi il for di sopra si poteva anche scrivere:

```
(for k=0 ; k<100; k++)   p[k] = k*3.14 ;
```

L'effetto, è comunque:

Nota [2] Per quanto riguarda calloc , malloc e sizeof:

- sizeof non è una funzione, ma un operatore unario; il valore dell'espressione sizeof(<tipo>) è il numero di byte che servono per una cella per <tipo> ; useremo la forma sizeof(<tipo>) ma non è l'unica possibile;
- calloc è una funzione a due argomenti: "quante celle" , "dimensione delle celle" ; il risultato è un puntatore, il cui valore è

  l'indirizzo della prima cella se la richiesta è andata a buon fine, NULL altrimenti;

scrivendo p = calloc (...); assegniamo a p questo valore.

Sostanzialmente, p = calloc (1,...); equivale a new(p);

- alcuni compilatori richiedono la forma pun = (* <tipo>) calloc (...); se pun è un puntatore_a_<tipo>
  [tecnicamente, (* <tipo>) è una operazione di *cast*, o conversione esplicita di tipo; serve per forzare il tipo del risultato di calloc a coincidere con quello di pun].
- usando **malloc** avremmo scritto: p = malloc (100 * sizeof(float));
  per quanto riguarda la sintassi, malloc è del tutto analoga a calloc solo che ha un unico argomento; per quanto riguarda l'effetto: calloc fornisce celle inizializzate a zero, mentre malloc non fa nessuna inizializzazione.

Nota [3] Per quanto riguarda free(p), valgono i discorsi fatti prima relativamente a dispose(p); il meccanismo è intelligente solo relativamente al numero di celle liberate, che sono tante quante ne erano state allocate con le corrispondente chiamata a calloc o malloc ; per il resto è compito del programmatore evitare guai; in particolare non si deve cercare di applicare free ad un puntatore che punta a qualcosa che non è stato ottenuto tramite calloc o malloc.

Array dinamici.

Da quello che abbiamo visto, si capisce che in C è possibile dimensionare gli array a *run time*, piuttosto che essere costretti a farlo a *compile time*.

Facciamo il solito esempio:

in input abbiamo n (intero >0) ed n razionali $a_1, a_2, ..., a_n$

vogliamo caricare gli a_j in un array aa e poi .....

Schema seguito finora:

```
#define MAX  100
.....
float aa[MAX] ;
int n ;
.....  /* mettiamo gli aj in aa  da 0 a n-1 */
.....  /* eccetera */
```

Supponendo che l'input sia corretto e dato nella forma

```
n                    su una riga
a1  a2  a3           su una o piu` righe, separati da spazi
```

la soluzione che ora possiamo adottare, si può schematizzare come segue.

```
.....
float  *  aa ;
int n ;
int k ;    /* indice ausiliario */
.....
scanf( "%d\n" , &n ) ;
aa = calloc ( n , sizeof(float) );
for (k=0; k<n; k++)
      scanf( "%f" , aa+k )       /* ma anche: scanf("%f" , &aa[k]) */
.....
for (k=0; k<n; k++)
      printf("%f\n" , *(aa+k))   /* ma anche: printf("%f\n" , aa[k]) */
.....
```

Nota. È chiaro che questo si può fare se il primo elemento dell'input è il numero esatto degli elementi da leggere e mettere nell'array. Ci sono situazioni in cui non è così: i dati sono in un file, conosciamo il loro formato, ma non sappiamo quanti sono. Due le soluzioni possibili: leggere due volte il file, la prima per contare gli elementi; oppure, usare una *lista* al posto dell'array. Le liste verranno viste parlando di tipi di dato; per il momento supponiamo di sapere quanti sono i dati in input.

Mettendo assieme quanto sopra con il modo di gestire gli array come parametri, otteniamo una grande flessibilità. Per capirlo, rivediamo lo schema descritto in Sezione 16.5 a proposito della procedura selsort:

```
#include ......
#define MAX 100   /* la dimensione massima per l'array da ordinare */

void selsort( float x [ ], int dim );
                /*  ordina l'array  x  da  0  a  dim  (compresi)  */
main( )
    {   int n ; .............
        float a[MAX];       /*  l'array da ordinare                    */
        ..................... /*  lettura di  n  e degli  n  elementi .... */
        selsort (a,n);
        ...................  /*  stampa del risultato                   */
    }

void selsort( float x[ ], int dim )    { ..................... }
```

Ora possiamo migliorare schema, abolendo MAX e dichiarando a come puntatore:

```
#include ......

void selsort( float x [ ], int dim );

main( )
     {  int n ; .............
        float * a ;         /*  l'array da ordinare                    */
        ..................... /*  lettura di  n,  dimensionamento di a,
                               caricamento dei dati .... come visto sopra  */
        selsort (a,n);
        ...................  /*  stampa del risultato                   */
    }

void selsort( float x[ ], int dim )    { ..................... }
```

Tutto questo ci porta a semplificare il modo di presentare gli algoritmi in pseudo codice.

Scriveremo tranquillamente cose del tipo a : array [1 .. **n**] of ... sapendo che esiste un equivalente in C.

Nel caso di procedure/funzioni, ci permettiamo un ulteriore semplificazione; vediamola su un esempio: una funzione che preso un array di interi restituisce la somma degli elementi.

In pseudo codice (seguendo lo "stil novo"):

```
function  somma ( aa : array [1 .. n] of integer )
    {    var k , res : integer
         res ← o;
         per  k = 1, 2, ..., n : res ← res + aa[k]      il problema è  n   vedere sotto
         return (res)
    }
```

Per capire il problema, traduciamo in C, in modo meccanico:

```
int somma ( int aa[ ] )
    {  int k ;
       int res = 0 ;
       for ( k = 0 ; k < n ; k++) res = res + aa[k] ;
       return(res);
    }
```

Non funziona: già in fase di compilazione si ha un errore: **n** è un identificatore non dichiarato.

La versione corretta è:

```
int somma ( int aa[ ], int n )
    {   ......  /* tutto come sopra */ }
```

Quindi, la convenzione che adotteremo è la seguente: scrivendo aa : array [inf .. sup] of ....

per un parametro di una procedura o funzione, possiamo poi usare nel corpo della procedura sia inf che sup; però bisogna ricordare che è solo un modo conciso di scrivere le cose e che traducendo in C può essere necessario aggiungere parametri corrispondenti a inf e sup.

19. Funzioni e procedure come parametri

Una funzione / procedura può avere parametri che sono a loro volta funzioni / procedure.

Vediamo per primo un esempio semplice: una procedura che, preso un array aa di interi ed una funzione unaria f da interi in interi, modifica aa applicando f a ciascun elemento.

Per non complicarci la vita, utilizziamo array a dimensione fissa.

Versione in pseudo-codice:

```
const  M = .....
procedura  map (     aa : array [1 .. M] of integer       parametro IN-OUT
                          f :  funzione da integer a integer                    )
    {    k : integer
         per  k = 1, 2, ..., M :  aa[k] ← f( aa[k] )
    }
```

Quindi, se bb è un array che contiene b_1, ..., b_M allora dopo la chiamata map(bb, g) bb conterrà $g(b_1)$, ..., $g(b_M)$.

Per capire l'uso che si può fare di tale procedura, supponiamo di scrivere un programma che contiene, oltre alla dichiarazione di map, le seguenti dichiarazioni:

function quad (y : integer) : integer { return(y * y) }

function cubo (y : integer) : integer { return(y * y * y) }

bb : array [1 .. M] of integer (riempito da 1 a M)

Allora:

la chiamata map(bb, quad) eleva al quadrato tutti gli elementi da 1 a M;

la chiamata map(bb, cubo) eleva al cubo tutti gli elementi da 1 a M.

In C, il tutto prende la forma seguente:

```
#include <stdio.h>
#define M  100
int quad (int y);    /* eleva al quadrato */
int cubo (int y);    /* eleva al cubo     */

void map ( int aa[ ] , int f(int) );
              /*  notare come si specifica il parametro funzione */

main( )
 {
   int k;       /*   k e` un indice */
   int bb[M] ;
   ..... /* mettiamo i dati in  bb  nelle posizioni da 0 a M-1 */
   map (bb, quad);
   .....
   map (bb, cubo);
   .....
 }

int quad (int y) { return (y*y) ;}

int cubo(int y) { return (y*y*y) ;}


void map ( int aa[ ] , int f(int) )
 {
   int k;
   for (k=0; k<M; k++) aa[k] = f(aa[k]);
 }
```

Nota.

Dal punto di vista del programmatore, i parametri funzione/procedura sono sempre parametri IN, in quanto interessa il loro "valore" (cioè quello che fanno) e non c'è nessuna idea di modificarli. A livello di implementazione, quello che viene passato alla procedura è l'indirizzo del codice, cioè un puntatore.

Vediamo ora un esempio più significativo.

La procedura selsort, vista in Sezione 16.3.2 e Sezione 16.5, può ordinare solo array di razionali. È facile modificarla per trattare il caso di elementi interi o caratteri: basta cambiare il tipo del parametro x e quello della variabile aux; tutto il resto va bene.

Se pensiamo di modificare la procedura per ordinare array in cui elementi sono record, allora i cambiamenti di sopra non sono sufficienti, bisogna modificare anche il corpo delle procedura; in particolare il test x[j] < x[imin].

In effetti, prima ancora di modificare il codice, dobbiamo chiederci come si possono ordinare degli elementi che sono record.

Per concretezza, prendiamo il caso di record a due campi: un campo matricola di tipo intero ed un campo cognome di tipo stringa. Possiamo ordinare questi record in tanti modi: solo in base al campo matricola (in modo ovvio), solo in base al cognome (ad esempio usando l'ordine lessicografico, cioè quello usato nei vocabolari), oppure combinando in qualche modo i due criteri.

A questo punto ci rendiamo conto che sarebbe bello poter scrivere una sola procedura che sia in grado di gestire i vari casi. Effettivamente, si può fare e nel modo seguente:

- si aggiunge un parametro funzione a valori booleani, chiamiamola confronto
- nel corpo della procedura selsort si sostituisce x[j] < x[imin] con confronto(x[j] , x[imin])
- al momento delle chiamata si passa a selsort, al posto di confronto , la funzione che usa il criterio di ordinamento che ci interessa in quel momento.

A questo punto viene voglia di fare un passo ulteriore: scrivere una procedura che sia in grado di ordinare array di record, indipendentemente dal particolare tipo di record, anzi: in grado di ordinare array con elementi di qualunque tipo.

Per fare questo bisognerebbe parametrizzare la procedura rispetto al tipo degli elementi dell'array; questo si può realizzare in C ed in Pascal in modo un po' primitivo, con una dichiarazione di tipo:

type Elem = **.....** oppure typedef **.....** Elem

Naturalmente, siamo costretti a scrivere qualcosa al posto di **.....** ; il punto è che concentriamo qui la specifica del tipo; se vogliamo cambiare tipo, dobbiamo modificare solo la definizione di Elem.

Ultimo problema: le assegnazioni che ci sono nella fase di scambi. In C, se w e z sono due variabili record (ma anche union) dello stesso tipo, si può scrivere l'assegnazione w = z (e si ottiene il risultato voluto), mentre in Pascal è vietato. Però anche in C siamo nei guai se proviamo a copiare un array in un altro (questo avviene se l'array da ordinare è un array di array).

Per toglierci dai guai, usiamo un ulteriore parametro per selsort: una procedura, chiamiamola scambia , che si occupi di fare lo scambio.

Mettiamo tutto assieme in un programma scritto in quasi-Pascal; il tipo Elem è quello che corrisponde ai record a due campi descritti prima, inoltre usiamo array a dimensione fissa; nella versione C passeremo agli array dinamici.

La nuova procedura di ordinamento la chiamiamo selsortall perché è in grado di ordinare array di qualunque tipo.

```
const   MAX = 100       dimensione degli array
        MS = 15         lunghezza massima delle stringhe cognome
type    Stringa = array [1 .. MS ] of char
        Elem = record   matricola : integer
                        cognome : stringa   end
        Vettore = array [1 .. MAX ] of Elem
var     bb :  Vettore       l'array da ordinare
        n : integer         la dimensione vera di  bb
        .....               altre variabili che servono .....

procedura selsortall (  x : Vettore             parametro IN-OUT
                        dim: integer            parametro IN
                        confronto :  funzione da Elem per Elem in boolean
                        scambia :  procedura con due parametri di tipo Elem    )
```

```
{   var     k2, j, imin : integer
    per  k2  = 1, 2, ..., dim-1  :      {
            trova minimo in  x  da k2 a dim; chiamiamo   imin   l'indice di questa posizione:
                imin ← k2
                per  j = k2+1, k2+2, ..., dim  :    if   confronto( x[j] , x[imin] )  then  imin ← j
                                        qui:  confronto( x[j] , x[imin] )    equivale a   x[j] <  x[imin]
            fase di scambio:
                scambia ( x[k2]  ,  x[imin] )
            }
}       fine di selsortall

function conf_1 (x, y : Elem) : boolean                 confronto di matricole
            {     return (x.matricola < y.matricola)    }

function conf_2 (x, y : Elem) : boolean                 confronto lessicografico
qui supponiamo che i cognomi corti siano completati aggiungendo degli spazi bianchi a destra e
sfruttiamo il fatto che nell'ordine standard dei caratteri lo spazio è minore di ogni lettera
            {   var  aux : boolean  ;   k : integer
                aux ← true  ;   k ← 1
                while (aux  and  k ≤ MS)  do
                    {   if   x.cognome[k] > y.cognome[k]  then  aux ← false
                        k ← k + 1
                    }
                return(aux)
            }

procedura  scambia ( x, y : Elem          parametri IN-OUT )
            {   var      auxmat : integer  ;    auxcog : stringa  ;   k : integer
                auxmat ← x.matricola ;  x.matricola ← y.matricola ; y.matricola ← auxmat
                per k  = 1, 2,..., MS :  auxcog[k] ← x.cognome[k]
                per k  = 1, 2,..., MS :  x.cognome[k] ← y.cognome[k]
                per k  = 1, 2,..., MS :  y.cognome[k] ← auxcog[k]
            }
```

Istruzioni del programma:

leggi i valori per n e per gli elementi di bb

.....

selsortall (bb, n, conf_1, scambia) *ora bb è ordinato, ordinando gli elementi (in modo crescente) in base alla matricola*

.....

selsortall (bb, n, conf_2, scambia) *ora bb è ordinato, ordinando gli elementi (in modo crescente) in base al cognome*

.....

Ricapitolando

La procedura selsortall è un grado di ordinare qualunque tipo di array nel senso che, senza cambiare una virgola di selsortall, quando la inseriamo in un programma dove

si precisano i tipi Elem e Vettore

si definiscono una funzione di confronto, cf , ed una procedura di scambio, sc , opportune

si ha un array bb di tipo Vettore di dimensione effettiva n

la chiamata selsortall (bb, n, cf, sc) ordina bb nel modo voluto.

L'unica condizione sugli array, perchè tutto funzioni, è che gli indici siano interi e partano da 1.

Per completezza, vediamo anche una versione in C, riducendo al minimo i commenti e senza usare le facilitazioni che il C offre per lavorare con le stringhe, ma con bb che è un array dinamico.

```
#include <stdio.h>
#include <stdlib.h>    /*  per calloc, malloc,....   */


typedef char  Boolean ;
#define  TRUE  1
#define  FALSE 0

#define  MS   15      /* lunghezza massima delle stringhe cognome */

typedef  struct  {  int  matricola ;
                    char cognome[MS] ;
                 }  Elem ;

typedef Elem * Pun_Elem ;      /* i puntatori ad Elem, vedi procedura scambia */


void selsortall ( Elem x[ ] , int dim ,
                  Boolean confronto ( Elem, Elem ) ,
                  void scambia ( Pun_Elem , Pun_Elem )
                ) ;

Boolean conf_1 ( Elem x, Elem y ) ;    /* confronto di matricole */

Boolean conf_2 ( Elem x, Elem y ) ;    /* confronto lessicografico */

void  scambia ( Pun_Elem px, Pun_Elem py ) ;

main ( )  {
  Elem  * bb ;          /*  l'array da ordinare  */
  int n ;
  altre variabili che servono
  .....
  leggi il valore per   n
  dimensiona  bb
  leggi i valori per gli elementi di  bb
  .....
  selsortall ( bb, n, conf_1, scambia) ;
  .....
  selsortall ( bb, n, conf_2, scambia) ;
  .....
 }
```

```
void selsortall ( Elem x[ ] , int dim ,
              Boolean confronto ( Elem, Elem ) ,
              void scambia ( Pun_Elem , Pun_Elem )
            )
 {
  int k2, j, imin ;
  for ( k2 = 0 ; k2 < dim-1 ; k2++ )
        {  imin = k2 ;
          for ( j =  k2+1; j < dim ; j++ )
               if  ( confronto( x[j] , x[imin] ) )  imin = j ;

          scambia ( &x[k2]  ,  &x[imin] ) ;
        }
 }

Boolean conf_1 ( Elem x, Elem y )  { return (x.matricola < y.matricola) ; }

Boolean conf_2 ( Elem x, Elem y )
 {  int k;
   for (k = 0; k < MS ; k++)
       if ( x.cognome[k] > y.cognome[k] )   break ;
   return(k = MS+1) ;
        /*  infatti l'uscita "regolare" dal for si ha con k = MS+1  */
 }

void  scambia ( Pun_Elem px, Pun_Elem py )
 {  int auxmat ;
   char auxcog [MS] ;
   int k ;
   auxmat =  (* px).matricola ;
   (* px).matricola = (* py).matricola ;
   (* py).matricola = auxmat ;
   for ( k = 0 ; k < MS ; k++ )  auxcog[k]        = (* px).cognome[k] ;
   for ( k = 0 ; k < MS ; k++ )  (* px).cognome[k] = (* py).cognome[k] ;
   for ( k = 0 ; k < MS ; k++ )  (* py).cognome[k] = auxcog[k]        ;
 }
```

Quanto sopra è, più o meno, tutto quello che si può fare in C, volendo mettere tutto in un unico file. La leggibilità e la modularità si può migliorare usando più file. Diamo solo lo schema, rimandando alle lezioni dedicate al C il compito di chiarire il meccanismo che permette di usare più file.

Una divisione ragionevole usa 6 file:

- uno con dichiarazioni ed altro di "uso comune", ad esempio il tipo Boolean;
- uno con la dichiarazione del tipo Elem ed i prototipi delle funzioni di confronto e scambio;
- uno con il codice di queste funzioni
- uno per il prototipo della procedura selsortall;
- uno con il codice di selsortall;
- uno con il main.

Vediamo lo schema (notare come sono messi gli #include):

file comune.h

```
typedef char  Boolean ;
#define  TRUE  1
#define  FALSE 0
.....   /* altro, che pero` qui non serve   */
```

file elementi.h

```
#include "comune.h"

#define  MS   15    /* lunghezza massima delle stringhe cognome */

typedef  struct  {  int  matricola ;
                 char cognome[MS] ;
               }  Elem ;
typedef Elem * Pun_Elem ;

Boolean conf_1 ( Elem x, Elem y ) ;  /* confronto di matricole */

Boolean conf_2 ( Elem x, Elem y ) ;  /* confronto lessicograf. */

void  scambia ( Pun_Elem px, Pun_Elem py ) ;
```

file elementi.c

```
#include "elementi.h"

Boolean conf_1 ( Elem x, Elem y )
  { return (x.matricola < y.matricola) ; }

Boolean conf_2 ( Elem x, Elem y )
  {  int k;
     for (k = 0; k < MS ; k++)
        if ( x.cognome[k] > y.cognome[k] )   break ;
     return(k = MS+1) ;
  }

void  scambia ( Pun_Elem px, Pun_Elem py )
  {   /* corpo di scambia, come sopra  */ }
```

file selsort.h

```
#include "elementi.h"

void selsortall ( Elem x[ ] , int dim ,
              Boolean confronto ( Elem, Elem ) ,
              void scambia ( Pun_Elem , Pun_Elem )
       ) ;
```

file selsort.c

```
#include "selsort.h"

void selsortall ( Elem x[ ] , int dim ,
                  Boolean confronto ( Elem, Elem ) ,
                  void scambia ( Pun_Elem , Pun_Elem )
                )
 {
  int k2, j, imin ;
  for ( k2 = 0 ; k2 < dim-1 ; k2++ )
        {  imin = k2 ;
           for ( j =  k2+1; j < dim ; j++ )
                if  ( confronto(x[j] , x[imin]) )  imin = j ;

           scambia ( &x[k2]  ,  &x[imin] ) ;
        }
}
```

file main.c

```
#include "selsort.h"

#include <stdio>
#include <stdlib>

main ( )  {
  Elem  * bb ;          /*  l'array da ordinare  */
  int n ;
  altre variabili che servono
  .............................
  leggi il valore per   n
  dimensiona  bb
  leggi i valori per gli elementi di  bb
  ......................................
  selsortall ( bb, n, conf_1, scambia) ;
  ......................................
  selsortall ( bb, n, conf_2, scambia) ;
  ......................................
 }
```

Fine della 1ª puntata