Breve corso di C/C++

Maurizio Falcone

versione 3.7, Ottobre 2009

Corso di "Laboratorio di Programmazione e Calcolo"
Laurea Triennale in Matematica
SAPIENZA - Università di Roma
A.A. 2009-10

"How do we convince people that
in programming simplicity and clarity -
in short: what mathematicians call 'elegance'-
are not a dispensable luxury,
but a crucial matter that decide between
success and failure?"

"Come convincere la gente che,
nella programmazione, semplicità e chiarezza -
in breve: quello che i matematici chiamano 'eleganza' -
non sono un lusso ,
ma un fatto cruciale che fa la differenza
tra successo e insuccesso?"

(E.W. Dijkstra, Selected writings on computing:
a personal perspective,1982)

Introduzione

Come hanno scritto B.W. Kernighan e D.M. Ritchie, nel primo libro di presentazione del linguaggio, "Il C è un linguaggio di programmazione di applicazione generale che presenta come proprie caratteristiche la sinteticità delle espressioni, moderne strutture di controllo e di dati, e un esteso insieme di operatori".

Il C è stato originariamente progettato ed implementato da Dennis Ritchie per il sistema operativo UNIX alla fine degli anni 70.

Caratteristiche di C

- Il C è un linguaggio procedurale moderno che permette una notazione molto compatta ed efficace

- E' portabile su molte architetture

- Permette al programmatore un controllo molto elevato sulla memoria della macchina, consentendo di ottimizzare il codice

- E' molto versatile *(multi-purpose)* e viene correntemente utilizzato per scrivere sistemi operativi, librerie scientifiche e grafiche oltre a molti altri pacchetti applicativi.

- Insieme alla sua versione C++ orientata alla programmazione ad oggetti, è uno dei linguaggi più diffusi nel mondo del calcolo scientifico e nell'industria.

Altri linguaggi sono più adatti a risolvere problemi in specifiche aree di applicazione:

Linguaggio	*Applicazione tipica*
Fortran 77 e *Fortran 90*	calcolo scientifico
Pascal	insegnamento
APL	calcolo simbolico, calcolo scientifico
Cobol	contabilità, gestione
Assembler	primitive di librerie (grafiche ad esempio)
DB3	gestione di banche dati (data-base)
BASIC	giochi
VISUAL BASIC	giochi, calcolo, gestione

Simboli

I simboli utilizzati dal linguaggio C sono

Lettere: da A a Z, da a a z

Numeri: 0, 1, 2, ..., 9

Simboli speciali: + - * / = ^ <> ( ) [ ] {} . , : ; ’ ! @ $ %|

C fa distinzione tra maiuscole e minuscole, cioè NON è vero che

PLUTO=PlUtO=PLUto=pLUTO=pluto

E’ pratica comune nel C usare le minuscole per i nomi delle variabili e delle funzioni/sottoprogrammi e le maiuscole per le costanti simboliche.

Bit e byte

I chip di memoria contengono *bits* di informazione.

Un bit ha unicamente due valori possibili, 0 oppure 1. Questa forte limitazione è superata dal fatto che un insieme di bit contigui può essere utilizzato per distinguere moltissime possibilità.

Ad esempio, 8 bits (= 1 *byte*) possono distinguere tra 2^8=256 differenti casi (sono tutte le possibili combinazioni di 8 valori 0 e 1). Il numero di queste possibilità è sufficiente a individuare ogni carattere presente sulla tastiera del vostro computer.

Un insieme di bytes, che siano contigui in memoria, può essere utilizzato per descrivere un numero, intero o in virgola mobile.

Bit e byte

In C e C++, ogni insieme di bytes in memoria per essere usato dal vostro programma deve essere dichiarato ed assegnato ad un tipo di dato (*data type*). Il tipo di dato scelto indica in che modo i bits devono essere interpretati per individuare il numero, il carattere o un altro tipo di dato.

Ogni byte di memoria ha un posizione precisa nel chip di memoria, tale posizione è individuata da un numero (il suo *indirizzo*). Questo indirizzo non va confuso con il contenuto dell'informazione contenuta nel byte (che è il suo *valore*).

C e C++ permettono di accedere sia all'indirizzo che al valore di un byte o di un insieme di bytes contingui in memoria.

Tipi di Variabili

In C le variabili devono sempre essere dichiarate prima di essere utilizzate, questa dichiarazione avviene all'inizio del programma o di ogni funzione/sottoprogramma (in C vengono chiamate *function*).

Per semplificare la loro definizione, il linguaggio C offre alcuni tipi di variabili predefiniti.
Ad esempio, sono predefiniti:

i numeri interi
i numeri reali
i caratteri

Per alcuni di questi tipi esistono due versioni, **short** e **long** (versione ridotta e versione estesa).

Convenzione: in questo manuale le parole scritte in **grassetto** sono comandi o parole riservate del linguaggio C.

int

Occupazione di memoria : 2/4 bytes (=16/32 bit per short/long)
Ogni intero n, tale che -32768 < n < +32767
Se l'intero è fuori da quell'intervallo si verifica un integer overflow.

I qualificatori short e long si applicano a int e corrispondono a due lunghezze diverse di interi (in termini di cifre).
Vengono interpretati in modo diverso a seconda della macchina e del compilatore, l'unica certezza è che short int ha meno cifre di long int.

In molti casi una delle due versioni coincide con il tipo int, ad esempio in Turbo C int=short int mentre per il compilatore Dev C++, int= long int.
Un numero long int occupa 4 bytes (= 32 bits), ed ha le limitazioni

$$-2.147.483.648 < n < 2.147.483.647$$

Un altro qualificatore per gli interi è unsigned, gli interi unsigned sono sempre positivi.

Esempio

```
int i;
long int n;
unsigned int m;
```

corrispondono alle dichiarazioni di un intero (i), un intero esteso (n) e di un intero positivo (m).

Nota importante: *si osservi che il ; è un separatore tra comandi C, che possono essere messi anche sulla stessa riga* (int i;long int n; unsigned int m;)

L'overflow intero *(integer overflow)* non viene dichiarato ma il risultato non è quello voluto.

Esempio

L'operazione

```
1000*100/50
```

non dà come risultato 2000 ma dà integer overflow (cioè 32768) perché le operazioni vengono eseguite da sinistra a destra. Le parentesi invertono l'ordine di esecuzione, dunque

```
1000*(100/50)
```

fornisce il risultato corretto.

float

Occupazione di memoria : 4 bytes

Sono gli x reali, tali che

$$3.4\ E\text{-}38 < |x| < 3.4\ E+38$$

e la mantissa nella rappresentazione in virgola mobile ha fino ad 8 cifre significative (semplice precisione).

Se si supera la limitazione superiore, viene segnalato un *errore di overflow*.

Se invece $|x| < 3.4\ E\text{-}38$ si verifica un *errore di underflow* e ad x viene sostituito 0.

double

Occupazione di memoria : 8 bytes

Sono gli x reali tali che

$$1.7\ E\text{-}308 < |x| < 1.7E+308$$

e la mantissa ha fino ad 16 cifre significative (doppia precisione). L'errore di underflow scatta evidentemente al di sotto di $1.7\ E\text{-}308$.

Attenzione!

Se sui reali si verifica un errore di:

overflow	causa l'arresto del programma
underflow	il risultato viene posto = 0, ma il programma prosegue l'esecuzione.

Esempio (in float)

a=exp(100)	*va in overflow con arresto del programma (runtime error)*
b=exp(-100)	*va in underflow, il programma continua l'esecuzione*

TABELLA RIASSUNTIVA

TIPO	*DIMENSIONE (in bit)*	*INTERVALLO*
int	16	da -32.768 a 32.767
unsigned int	16	da 0 a 65535
long int	32	da -2.147.483.648 a 2.147.483.647
float	32	3.4 E-38<|x|< 3.4 E38
double	64	1.7 E-308<|x|<1.7 E+308

char

Occupazione di memoria: 1 byte

Carattere nell'ambito dei caratteri ascii (sono 255).

Anche le *stringhe* (=sequenze di caratteri e spazi bianchi) sono variabili di tipo char

Ad esempio, si può assegnare una intera frase mettendola tra apici ad una variabile di tipo char

Esempio

"Il risultato dell'integrale e':"

"La soluzione cercata e'"

Nota

In C il tipo booleano non è predefinito. Di solito una variabile booleana viene sostituita con una variabile intera che può valere solo 1 (true) oppure 0 (false).

In C++ invece esiste il tipo **bool ,** a variabili di questo tipo si applicano solitamente gli operatori logici (AND, OR, NOT,..)

Conversioni di tipo (cast)

Quando vengono eseguite operazioni tra variabili di tipo diverso, ad esempio una moltiplicazione tra un intero ed un reale, il C le tratta come se fossero tutte dello stesso tipo, considerandole implicitamente tutte del tipo "superiore" (quello tra i due che ha maggiore occupazione di memoria).

In particolare, nelle espressioni aritmetiche la sequenza delle trasformazioni di tipo viene eseguita esattamente in questo ordine:

1. short int viene convertito in int, float viene convertito in double;

2. se uno dei due operandi è di tipo double, anche l'altro viene convertito in double e il risultato dell'operazione è un double;

3. se uno dei due operandi è di tipo long, anche l'altro viene convertito in long e il risultato dell'operazione è long;

4. se uno dei due operandi è di tipo unsigned, anche l'altro viene convertito in unsigned e il risultato dell'operazione è unsigned;

Nei casi in cui la conversione di tipo non venga effettuata implicitamente dal compilatore oppure quando si vuole evitare un risultato indesiderato si può forzare una conversione di tipo (cast) mettendo tra ( ) il tipo desiderato prima del nome della variabile.

Esempio

```
(float) media/n  /* con media ed n int*/
(double) minimo+i*dx /*con minimo, dx float e i int*/
```

impongono il tipo float al valore della divisione media/n ed il tipo double al risultato dell'espressione minimo+i*dx.

Un caso tipico in cui occorre imporre il tipo del risultato è nella divisione di due numeri interi. Infatti, se non venisse fatto un cast il risultato della divisione sarebbe intero e si perderebbero tutte le cifre dopo la virgola.

Esempio

```
#include <stdlib.h>
#include<stdio.h>

int main (void) {
        int i,n,x,somma;
        float media;
printf("Quanti numeri vuoi inserire?");
scanf("%d",&n);
printf("Inserisci  %d  numeri interi: ", n);
        for (i=0;i<n;i++) {
                scanf("%d",&x);
                somma=somma+x;
                }
media=(float)somma/n;
printf("La media dei numeri inseriti e' %f \n", media);
system("pause"); /*sui PC serve a bloccare il video*/
return(1);
}
```

Puntatori

Le variabili possono anche essere di tipo *puntatore*.

Una variabile di tipo puntatore conterrà l'indirizzo di memoria di un'altra variabile di tipo int, char, float,...

Non è possibile definire una variabile puntatore "universale", cioè che "punti" indifferentemente ad una variabile intera, reale, character,... Di conseguenza, *a fianco del puntatore bisogna sempre indicare il tipo di variabile cui "punta"*.

Per indicare che una variabile è un puntatore, nella sua dichiarazione occorre mettere * prima del nome.

Esempio

```
int *pn,*pm;
float *px,*py, x;
```

definiscono rispettivamente 2 puntatori ad interi (pn,pm), 2 puntatori a reali (px,py) e 1 variabile reale x.

Ci sono due operatori particolarmente importanti per lavorare con i puntatori: & e *.

L'operatore & applicato ad una variabile restituisce il suo indirizzo di memoria (puntatore alla variabile).

L'operatore * applicato ad un indirizzo di memoria (puntatore) restituisce il valore contenuto in quell'indirizzo.

Attenzione: alcuni comandi di C (come ad esempio scanf) richiedono che sia indicata una variabile puntatore

Esempio

scanf("%d", &m);	*legge da tastiera l'intero m (che si trova all'indirizzo di memoria &m)*

Identificatori (nomi)

Sono i nomi relativi a costanti, tipi di variabili, variabili e funzioni, devono essere costituiti da una lettera seguita da una qualunque combinazione di lettere, cifre o sottolineature (cioè un nome corrisponde ad una stringa di caratteri).
Un nome può contenere fino a 127 caratteri tutti significativi.

Esempio

Totale	Pari	P3	estremo_b
somma	prodottoxy		pix

3RD *non va bene: inizia con un numero*

D2 grafico *non va bene: c'è uno spazio bianco*

raggio% *non va bene: c'è un carattere speciale*

Per mantenere una certa trasparenza nella scrittura del programma NON risparmiate caratteri sui nomi.

Date alle variabili dei nomi CHIARI che indichino il loro contenuto, stabilite le vostre convenzioni e ricordate che il C distingue tra MAIUSCOLE e minuscole .

Esempio

i,j,k,l,m,n	*indicano di solito variabili intere*
x,y,z,t	*indicano di solito variabili reali (diverse da X,Y,Z,T)*

Una vecchia convenzione (sempre utile) è quella di indicare una variabile intera con un nome che inizi con una delle lettere i,j,k,l,m,n

E' meglio evitare di usare i nomi `x,y,z` per variabili reali che devono servire in varie funzioni/sottoprogrammi perchè è difficile ricordare il loro significato quando si rilegge il programma.

Meglio utilizzare almeno 3 caratteri per i nomi delle variabili, ad esempio

`x_max`	`ymax`
`minimo`	`colore`
`matriceA`	`vettoreX`
`somma`	`mat_B`

sono buoni nomi di variabili.

Parole riservate

Ci sono alcune *parole riservate* che C utilizza e che *non* possono essere utilizzate dall'utente come nomi di variabili o di funzioni. Queste parole devono essere scritte in minuscolo e corrispondono a *istruzioni del linguaggio C*.

Sono riservate le parole corrispondenti a comandi C, come ad esempio:

main	**define**	**for**	**if**
then	**else**	**while**	**float**
int	**double**	**char**	**do**

ed alle funzioni matematiche, ad esempio:

sin	**cos**	**abs**	**log**

E' bene limitare la lunghezza di una linea di programma ad un centinaio di caratteri, in modo da poter leggere il programma comodamente.

NUMERI

Un reale (float, double) può essere scritto in virgola mobile oppure no, sia in input che in output. Nel caso dell'output viene sempre scritto in virgola mobile a meno che non venga specificato un formato di scrittura diverso.

Esempio

-6.352469E+11 1.274E+2 127.4

sono scritture corrette (le ultime due sono equivalenti)

Attenzione

1. *viene interpretato come un reale*
1 *viene interpretato come un intero*.

Un intero può essere sempre sostituito ad un reale ma non è (ovviamente) vero il viceversa.

Bisogna fare attenzione perché alcune funzioni predefinite in C accettano solo argomenti interi (es. la funzione fmod(n,m) che dà il resto della divisione intera tra n ed m) .

STRINGHE

Sono sequenze di caratteri ASCII comprese tra “ ”

Esempi

“Sono una stringa”

“Massimo: ”

“Inserisci il valore del parametro p: ”

COMMENTI

Tutto ciò che è contenuto tra /* */ viene considerato come un commento dal compilatore C e non viene tradotto in linguaggio macchina (quindi dal punto di vista della esecuzione non influenza il risultato del programma).

In C++, // commenta il testo che segue fino a fine riga.

Esempi

```
/*Calcolo dei valori della funzione */
/*Stampa */
/* Questo e' il calcolo della funzione */
x+y; // somma delle variabili
```

Per escludere alcune righe di programma dalla esecuzione basta quindi metterle tra /* */.
Questi commenti possono servire ad indicare cosa fa una parte del programma oppure si possono usare per eliminare (temporaneamente) una parte del programma che dà errori in compilazione per vedere se il resto del programma gira.

Esempio

```
#include <stdlib.h>
#include<stdio.h>
int main(void){
   int i,n,m, x, somma,prodotto; float media;
printf("Quanti numeri vuoi inserire?"); scanf("%d",&n);
printf("Inserisci %d numeri interi: ", n);
somma=0;prodotto=1;
        for (i=0;i<n;i++) {
              scanf("%d",&x);
        /*    somma=somma+y;
              prodotto=prodotto*y;  parte eliminata */
              }
media=(float)somma/n;
printf("La media dei numeri inseriti e' %f \n", media);
system("pause"); /*sui PC serve a bloccare il video*/
return(1);}
```

Struttura di un programma C/C++

Un programma C è articolato in sottoprogrammi che vengono chiamati funzioni, dunque una *funzione* è un blocco di istruzioni destinato a risolvere uno specifico sottoproblema. In una funzione entrano alcune variabili (dati in ingresso) ed escono i risultati (dati in uscita).
Non c'è correlazione tra il tipo dei dati in ingresso e quello dei dati in uscita. I risultati non sono necessariamente valori numerici.

In C le procedure (tipiche di Pascal) e le subroutine (tipiche di Fortran) vengono chiamate *funzioni* indipendentemente dal fatto che producano o meno un risultato numerico.
Il programma C viene preceduto dai comandi del pre-processore (facilmente individuabili perchè iniziano con #).

```
#comando_1 per il pre-compilatore
#comando_2 per il pre-compilatore
                .
                .
#comando_n per il pre-compilatore

        Blocco di dichiarazione;
        Funzione_1(argomento1, argomento2, ....);
        Funzione_2(argomento1, argomento2, ....);
        Funzione_3(argomento1, argomento2, ....);
                .
                .
                .
main (argomento1, argomento2, ....)
        {
            Blocco di dichiarazione;
            Blocco di istruzioni;
                                    }
```

Le variabili definite nel primo blocco di dichiarazione sono riconosciute da tutte le funzioni.

main è la funzione "programma principale", tra parentesi () ci possono essere degli argomenti mentre tra {} sono indicate le istruzioni o le chiamate alle funzioni.

Ogni funzione/sottoprogramma deve essere definita prima di poter essere chiamata dal main o da altre funzioni.

Gli argomenti tra () non sono obbligatori ma le () si.

Di solito una funzione prende degli argomenti e restituisce un risultato. Poichè sia gli argomenti che il risultato sono opzionali è possibile che una funzione non abbia argomenti e/o non fornisca un risultato. Se una funzione non ha argomenti si può indicare indifferentemente come

```
nome_funzione ()
nome_funzione (void)
```

Esempio

```
int massimo (int m,n)
float potenza (float x, int n)
void stampa (int *pa)
void scrivi (void)
```

La funzione massimo usa due variabili intere (m,n) e fornisce un valore intero.

La funzione potenza usa una variabile reale (x) e una variabile intera (n) e fornisce un valore reale.

La funziona stampa usa un puntatore ad una variabile intera (pa) e non fornisce nessun risultato (void).
La funzione scrivi non ha variabili (void) e non fornisce nessun risultato (void).

E' importante sapere che il programma viene eseguito a partire dalla parola chiave main che caratterizza la funzione "programma principale".

La funzione main è una funzione come tutte le altre, l'unica sua particolarità è che deve esistere nel testo di ogni programma. Viene solitamente messa alla fine, come ultima funzione. Se si vuole mettere invece il main all'inizio del programma occorre definire le funzioni che verranno usate, dopo i comandi del pre-compilatore e prima del main
Le funzioni vanno indicate specificando il loro *prototipo*

tipo_di risultato nome_funzione (argomenti)

Esempio

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>


/* lista funzioni*/
double g(double,int);
double c(double,int);
void leggi_intervallo(double &,double &);
void leggi_finetempo(double &);
void trova_passo(int,double,double,double &);
void leggi_tempo(int &);
void leggi_spazio(int &);
/* fine lista funzioni*/
int main()
{.....
return(1);
}

/*segue poi il dettaglio delle singole funzioni*/
```

Questa scelta è in genere più elegante e leggibile.

Di solito si assegna in uscita alla funzione main un valore intero (0 o 1) che serve solo ad indicare che il programma è stato eseguito.

E' bene ricordare che la stessa struttura viene riprodotta all'interno di ogni funzione, ricordiamola:

```
(tipo_di_risultato) nome_funzione(argomenti){
                    Blocco di dichiarazione;
                    Blocco di istruzioni;
}
```

Esempio

```
double g(double x,int c3){
double y;
if((x>=-1)&&(x<=1))
            {if(c3==1) y=1-x*x;
            if(c3==2) y=1-fabs(x);
            if(c3==3) y=1;
            if(c3==4) y=cos(2*atan(1)*x);}
else
            y=0;
return y;
}
```

Il pre-compilatore C

Il compilatore C contiene un pre-compilatore che serve a definire le costanti, richiamare le librerie ed includere file esterni.

Le linee che iniziano con # indicano comandi indirizzati al preprocessore .

Vediamo alcune istruzioni tipiche:

1. Chiamata a librerie esterne (#include)
2. Dichiarazione delle costanti (#define)

Attenzione: i comandi del pre-compilatore non vanno chiusi da ; anzi la presenza di un ; è fonte di errori.

1. Chiamata a librerie/files esterni

Un'istruzione del tipo

#include "nome_file"

causa la sostituzione di quella riga con l'intero contenuto del file nome_file. Il file viene ricercato nella directory del file sorgente e in altre directory standard (che vengono solitamente gestite dal sistema operativo).

Se il nome-file appare tra < >, la ricerca viene effettuata solo sulle directory standard (definite durante l'installazione del compilatore C).

Esempi

```
#include <stdlib.h> /*include la libreria standard */
#include <math.h> /*include la libreria matematica*/
#include <stdio.h> /*include la libreria standard di i/o*/
#include <iostream.h> /*include altre funzioni i/o*/
#include <conio.h>/*include la libreria di controllo i/o*/
#include <graphics.h> /*include la libreria grafica*/
```

dove i/o sta per input/output (ingresso/uscita).

La libreria stdio.h gestisce la stampa e la lettura dei dati (ad esempio, le istruzioni scanf e printf sono in questa libreria)

La libreria iostream.h gestisce altri comandi per la stampa e la lettura dei dati (ad esempio, le istruzioni cin >> e cout << sono in questa libreria)

La libreria conio.h gestisce i controlli della stampa/lettura dei dati (ad esempio, il comando clrscn che pulisce il video appartiene a questa libreria) e della posizione del cursore.

La libreria math.h contiene tutte le funzioni matematiche pre-definite in C (sono moltissime).

Ulteriori informazioni sui files .h (*headers*) disponibili sono contenute nell'help in linea e/o sul manuale del compilatore.

2. Costanti

Nel linguaggio C le costanti vengono definite per prime e vengono valutate durante la compilazione invece che in esecuzione. Per i nomi delle costanti si usano di solito caratteri MAIUSCOLI.

Esempio

```
#define LIMIT  150
#define MIN   25.1
#define NOME  “Pluto”
```

Definisce LIMIT pari a 150, MIN pari a 25.1 e NOME uguale alla stringa “Pluto”.

Sono predefinite le costanti

```
Pi = 3.1415926536
False=0
True=1
Maxint = 32767
```

Il comando

```
#undefine nome_costante
```

cancella la definizione relativa a nome_costante.

Esempio

```
#undefine LIMIT
#undefine MIN
#undefine NOME
```

cancellano le assegnazioni dell'esempio precedente.

Blocco di dichiarazione

Nel blocco di dichiarazione relativo alla fase di compilazione vengono definiti i nuovi tipi di variabili e vengono assegnate le variabili ai vari tipi. La definizione dei tipi di variabile in C è molto flessibile, qui citiamo solo alcune delle possibilità: *tipi enumerativi* (enum), *tipi strutturati* (struct) e *ridefinizione di tipo* (typedef).

Esempi di dichiarazione di tipi di variabili

```
enum day  {lun, mar, mer, gio, ven, sab,dom};
/*definizione del tipo enumerativo day
seguito tra { } dall'insieme dei suoi valori*/

struct persona
{ char nome[20];
  char cognome [20];
  int giorno_nascita, mese_nascita, anno_nascita};
```

```
typedef  int vettore[10];
/*definizione del tipo vettore come un vettore a
10 componenti intere*/

typedef float matrice[10][10] ;
/*definizione del tipo matrice come una matrice
a valori reali 10 per 10*/
```

L'istruzione enum definisce il tipo day, le variabili di questo tipo potranno assumere unicamente i valori indicati (lun, mar, mer, gio, ven, sab,dom).

L'istruzione struct definisce invece il tipo strutturato persona che sarà costituito da due stringhe di massimo 20 caratteri (nome e cognome) e da tre numeri interi (corrispondenti alla data di nascita).

L'istruzione typedef *non crea un nuovo tipo* ma rinomina un tipo già esistente in modo che sia più semplice e chiara la definizione delle variabili.

Nei *tipi strutturati* una singola variabile è costituita da un insieme *ordinato* di variabili eventualmente *di tipo diverso*. E' possibile che qualcuna delle variabili utilizzate all'interno della definizione sia di tipo puntatore.

Dal momento che è possibile definire dei nuovi tipi di variabile è anche possibile definire un tipo-puntatore associato al tipo di variabile con il comando

typedef <Tipo _dato> *<Tipo_puntatore>

Esempio

typedef **struct** persona *punta_persona;

Assegnazioni delle variabili

Ogni variabile va assegnata ad un tipo prima di essere utilizzata. Il tipo viene indicato prima del nome della variabile.

Esempio

```
float risultato, somma, x,y,*px,*py;
int  i, j, k, minimo,*pminimo;
double derivata;
day giorno ;
```

la dichiarazione precedente definisce

come reali (semplice precisione) le variabili risultato, somma, x, y

come puntatori reali le variabili px e py

come interi le variabili i,j,k,minimo

come puntatore intero la variabile pminimo

come reale (doppia precisione) la variabile derivata

come day la variabile giorno

Variabili globali, variabili locali

Tutte le variabili devono essere definite prima di poter essere utilizzate.

Se sono definite nel blocco di dichiarazione iniziale sono riconosciute in tutte le procedure del programma *(variabili globali)* altrimenti valgono solo all'interno della funzione/ sottoprogramma nel quale sono state dichiarate *(variabili locali)*.

La definizione di una variabile locale cancella quella globale con lo stesso nome.

Per assegnare un valore ad una variabile si usa, come al solito, il simbolo **=**

Esempio

```
max=min+altezza;
prodotto=a*b;
```

E' importante distinguere tra la definizione di una variabile (che indica solo di che tipo è)

```
float max;
```

dalla assegnazione del valore della variabile

```
max=min+alt;
```

(che assegna a `max` il valore dato dalla somma di `min+alt`).

Si possono definire vettori e matrici di variabili dello stesso tipo.

Esempio: definizioni di matrici e vettori

```
int  vet [10];
float matrix [10] [10] ;
double vet1[10], vet2[50],vet3[100];
```

In questo caso l'assegnazione dei valori della matrice e del vettore verrà fatto assegnando il valore a ciascuno degli elementi

```
matrix[0] [0]=2.3
matrix[0] [1]=3.72
      .....
matrix[9][9]=-43.14
```

***Nota**: in C vettori e matrici hanno come primo indice 0, dunque un vettore V a 10 componenti va da V[0] a V[9].*

Operatori

In C esistono molti operatori aritmetici e logici e alcuni di questi hanno due forme, una forma estesa ed una forma compatta.

Operatori aritmetici

+ (somma) , - (sottrazione), $*$ (prodotto), / (divisione),
% (modulo, cioè resto della divisione tra interi).

Operatore di assegnazione in forma estesa (=)

=

Esempio

```
x=a*b;
y=-23.41;
```

Operatori aritmetici di assegnazione in forma compatta

Sono : **++, --, +=, -=, ∗=, /=**

Questi operatori permettono di scrivere in forma compatta alcune tipiche assegnazioni e sono più efficienti (in esecuzione) della forma estesa.

Forma compatta	Forma estesa
x++	x=x+1
x--	x=x-1
x+=8	x=x+8
x-=10	x=x-10
x∗=5	x=x∗5
x/=2	x=x/2

La forma estesa spiega il significato di questi operatori e può essere utilizzata al loro posto.

Operatori logici di confronto

= =	uguale
!=	diverso
<	minore
>	maggiore
<=	minore o uguale
=>	maggiore o uguale

Esempio

```
x==y    /*x uguale ad y*/
x!=y    /*x diverso da y*/
x<=y    /*x minore o uguale ad y*/
```

Operatori logici (AND, OR, NOT)

In C, gli operatori logici hanno una sintassi particolare.

Sintassi	Significato
!	not
&&	and
||	or

```
! (condizione 1)
(condizione1)&& (condizione2)
(condizione1)||(condizione2)
```

Si applicano solitamente a variabili booleane, ma in C è anche possibile applicarli a valori numerici. La convenzione è che qualunque valore diverso da 0 è TRUE.

Esempio

```
!(x>1) /* equivale a x≤1 */
(x<4) && (x>0) /*  equivale a 0<x<4 */
(x<4) || (x>6.1) /* equivale a {x<4}U {x>6.1} */
```

Le espressioni logiche vengono valutate da sinistra a destra e la valutazione si interrompe non appena è possibile stabilire con esattezza il loro valore.

Le parentesi () servono a modificare l'ordine di priorità in esecuzione. Le condizioni tra le parentesi () più interne vengono valutate per prime.

Esempio

((condizione1) && (condizione2)) || (condizione3)

Nell'esempio, vengono valutate

per prime le condizioni 1 e 2

poi la condizione and (&&)

infine la condizione 3 e l'or (||)

Se (condizione 1) è falsa si passa direttamente a valutare (condizione 3) e l'or (||) perchè la condizione dell'and (&&) è certamente falsa.

Operatori relativi ai puntatori (&, *)

Ci sono due operatori particolarmente importanti per lavorare con i puntatori: & e *.

L'operatore & applicato ad una variabile restituisce il suo indirizzo di memoria (puntatore alla variabile).

L'operatore * applicato ad un indirizzo di memoria (puntatore) restituisce il valore contenuto in quell'indirizzo.

Se a è una variabile e pa il suo puntatore avremo quindi:

```
&a  restituisce pa
*pa restituisce a
```

Esempio
Il comando
&x
individua il puntatore corrispondente alla variabile x
mentre, fuori da un blocco di definizione, l'espressione

*pminimo
indica il valore della variabile puntata da pminimo

Ordine di priorità nelle operazioni

In ordine di priorità di esecuzione, le operazioni sono:

prima priorità	***, /**
seconda priorità	**+, -**

Le parentesi () servono ad invertire le priorità. L'esecuzione viene effettuata a partire dalle espressioni incluse tra le parentesi () più interne.

Gli operatori logici hanno priorità minima, cioè i confronti vengono eseguiti solo dopo che le altre operazioni (+, -, *, /) sono state effettuate.

Funzioni matematiche

Moltissime funzioni matematiche sono pre-definite in C, eccone alcune:

abs(x), fabs(x)	valore assoluto intero/reale
atan(x)	arco tangente
cos(x)	coseno
sin(x)	seno
log(x)	logaritmo naturale (in base e)
exp(x)	esponenziale in base e
sqrt(x)	radice quadrata
pow(x,y)	potenza di x con esponente reale y
floor(x)	troncamento all'intero inferiore
ceil(x)	arrotondamento per eccesso (all' intero superiore)

La potenza con esponente reale è una funzione della libreria matematica. Per una lista completa delle funzioni disponibili in C e per avere informazioni sul loro funzionamento consultare l'help in linea e/o il manuale del compilatore.

Esempi

```
fabs (-1.5) = 1.5
abs (1) = 1      /* intero */
fabs (1.) = 1.   /* reale */
floor(1.6)=1.
ceil(1.6)=2.
frac (1.5) = .5     /* frac (x) = x - floor (x) */
```

Le funzioni matematiche sono contenute nella libreria math.h che andrà quindi inclusa col comando del pre-compilatore

```
#include <math.h>
```

Funzione	Argomento	Valore
abs	intero	intero
fabs	reale	reale
atan	reale	reale
cos	reale	reale
sin	reale	reale
log	reale (>0)	reale
exp	reale	reale
sqrt	reale (≥0)	reale
pow	2 reali	reale
floor	reale	reale
ceil	reale	reale

Nota: *per forzare un valore intero in uscita occorre fare una conversione di tipo (cast).*

Una scelta errata dell'argomento produce come risultato nan (not a number).

Esempio: log(-1), sqrt(-1).

Funzioni di Input /Output

Le unità di default (cioè quelle utilizzate da C in assenza di vostre indicazioni specifiche) sono:

lo schermo per l'output

la tastiera per l'input

Il comando per leggere è

scanf ("%formato", indirizzo_della_variabile)

Mentre il comando per scrivere è

printf ("%formato1",...,"%formatoN", valore_della_variabile_1,... ,valore_della_variabile_N)

Le funzioni scanf e printf sono nella libreria stdio.h che gestisce la stampa e la lettura dei dati.

Solo in C++

è possibile utilizzare delle istruzioni semplificate che non hanno bisogno dell'indicazione del formato

```
cin>> nome variabile /*per leggere da tastiera */
cout<< stringa o nome variabile /*per scrivere a video*/
```

Esempio

```
cout<<"\ndammi un numero reale x=";
cin>>x;
y=fabs(x);
cout<<"\n\n modulo di "; cout<<x; cout<<" = ";
cout<<y;
```

Le istruzioni cin e cout sono contenute nella libreria iostream.h

scanf (“%formato”, indirizzo_della_variabile)

Permette di leggere da tastiera un dato con il formato prescelto. Il dato verrà memorizzato nell’indirizzo di variabile indicato.

Esempio

```
scanf (“%d”, &m)
/* legge un intero e lo memorizza nella variabile m*/

scanf (“%e”, &a)
/* legge un reale e lo memorizza nella variabile a*/

scanf (“%lf”, &b)
/* legge un reale doppia precisione e lo memorizza
nella variabile b*/
```

Si osservi che in tutti gli esempi l’operatore & di fronte al nome della variabile è essenziale per indicare l’indirizzo di memoria della variabile (senza & il comando è errato).

PRINCIPALI FORMATI DI INPUT/OUTPUT

%d	intero decimale
%ld	intero esteso (long int)
%u	intero decimale senza segno
%e	numero in virgola mobile
%f	numero in virgola mobile
%lf	numero in virgola mobile (double)
%c	carattere
%s	stringa di caratteri

La differenza tra il formato %e ed il formato %f è che 441.9 viene scritto come:

4.419000e+02 in formato %e

441.900000 in formato %f

printf (“%formato1”,...,“%formatoN”, valore_della_variabile_1,... ,valore_della_variabile_N)

Il comando permette di scrivere il contenuto di più variabili specificando formati diversi per ognuna delle variabili.

Esempi

```
printf (“%d”, m);
/*scrive il contenuto della variabile intera m*/

printf (“%d”,“%e”,“%lf”,m,a, b,);
/*scrive il contenuto delle variabili m,a, b*/

printf (“Valore massimo = %e\n”,max);
/*scrive la stringa “Valore massimo =”, il contenuto
della variabile max e va a capo (\n)  */
```

Nell'istruzione printf si possono inserire anche dei comandi di formattazione del testo quali

\t	tabulazione
\n	avanzamento di riga (line feed)
\f	cambio pagina (form feed)
\r	a capo (carriage return)
\a	segnale acustico (bell)

Nella scelta del formato si può anche indicare il numero dei caratteri destinati alla rappresentazione del numero

Esempio

```
%5d /* intero a 5 cifre*/
%4.3lf /* 4 cifre prima e 3 cifre dopo la virgola*/
```

Lettura/scrittura da/su file

In C e C++ è possibile leggere/scrivere da/su file.
Occorre aprire il file in lettura/scrittura, assegnargli un nome/numero per poter indirizzare i comandi di lettura/scrittura e alla fine chiudere il file.
La libreria stdio.h contiene i comandi necessari e va quindi inclusa con

```
#include <stdio.h>
```

le istruzioni più semplici per leggere/scrivere sui files sono:

```
fopen("nome_file","w") /*apertura in scrittura*/
fprintf(nomefile, formati,  argomenti)
 /*scrittura su file (precedentemente aperto)*/
fscanf(nomefile, formato, argomento)
  /*lettura da file (precedentemente aperto)*/
close(nome_file) /*chiusura*/
```

Esempio

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
FILE *file1;
FILE *file2;

main()
{
file1=fopen("file1.dat","w") /*apertura in scrittura di
file1*/
file2=fopen("file2.dat","w"); /*apertura in scrittura di
file2*/

.....
fprintf(file1,"%lf %lf\n",x[i],d[i][j]);
/*scrive su file1 due double e va a capo*/
.....
fprintf(file2,"%lf %lf\n",y[i],f[i][j]);
/*scrive su file2 due double e va a capo*/
......
fclose(file1);/*chiude file1 */

fclose(file2);/*chiude file2 */
}
```

E' molto comune utilizzare questi comandi per scrivere un file di dati per la grafica. Il file conterrà, ad esempio, i punti (x, f(x)) sul grafico della funzione f.
Il file di dati potrà poi essere visualizzato attraverso una libreria grafica (ad esempio GNUplot o MATLAB).

E' possibile anche lanciare la grafica dall'interno del programma C con il comando

```
system("startY:\\Gnuplot-3.7.1\\wgnupl32.exe
graf.txt");
```

dove `Y:\\Gnuplot-3.7.1` è il path della libreria GNUplot
e `graf.txt` è un file di comandi di GNUplot.

GNUplot è una libreria grafica public domain, per maggiori informazioni sui suoi comandi guardare il manuale che viene distribuito con la libreria.

Istruzioni composte

Un blocco di istruzioni comprese tra {} si chiama *istruzione composta*.

```
{istruzione1;
 istruzione2;
 istruzione3;
}
```

Esempio

```
void main (void) {
float a,somma;
int   b;
      a=3.21;b=5;
      somma = a+b;
      printf("Somma = %f \n",somma)
}
```

Produce su schemo:

La somma e': 0.821E+1

e manda a capo il cursore.

Cicli

Ci sono almeno tre modi diversi per costruire un ciclo in *C* e occorre utilizzare quello che meglio si adatta alle esigenze del problema che si vuole risolvere.

while (espressione/condizione) istruzione;

do istruzione; **while** (espressione/condizione);

for (variabile di controllo= espressione iniziale;
condizione di ciclo; legge di incremento)
istruzione;

Le espressioni possono essere espressioni logiche (condizioni) che hanno un valore booleano (vero/falso), ma possono anche essere espressioni qualsiasi. Nel secondo caso vengono considerate false quando il loro valore è zero, vere altrimenti.

Le istruzioni possono essere istruzioni composte (racchiuse tra {}).

Il ciclo do...while corrisponde al ciclo repeat...until del Pascal.

I cicli possono essere uno dentro l'altro (*nidificazione*). L'esecuzione di un break interrompe il ciclo più interno ma non gli altri.

1. **while** (condizione) istruzione;

esegue l'istruzione finché il valore della condizione è *true*, appena quel valore diventa *false* esce dal ciclo

Esempio

Calcola 1 + 1/2 + 1/3 + ... + 1/n

```
 /*programma per  while*/
int main (void) {
int n;
float h;
        scanf ("%d",&n);
        printf("%d",n);
        h= 0;
        while (n>0)
              {h= h + 1/n; /*oppure h+=1/n*/
                n = n-1; /* oppure n--*/
              }

        printf ("Somma=%e \n",h);
        return(1);
}
```

Produce (se n=10): 10

Somma = 0.292896E+1

2. **do** istruzione; **while** (condizione);

Ripete le istruzioni finché il valore della condizione è *true*. Poiché l'istruzione viene eseguita prima della valutazione della espressione, *il ciclo viene eseguito almeno una volta*.

Vediamo lo stesso esempio con la nuova costruzione

Esempio

```
/*programma  per do ... while*/
int main (void)  {
int n;
float h;
scanf ("%d",&n);
printf("%d",n);
            h=0;
            do{h = h+1/n;  /*oppure h+=1/n */
                n=n-1; /* oppure n-- */
                   }
                while (n!=0);
printf("Somma=%e \n",h);
return (1);
}
```

Il risultato è lo stesso ma il secondo programma non va bene se $n \leq 0$ perché,

se $n=0$ c'è un errore di division by zero

mentre

se $n<0$ il ciclo viene ripetuto infinite volte (visto che la variabile n in quel caso non può annullarsi mai).

3. **for** (variabile di controllo=espressione iniziale;
condizione di ciclo; legge di incremento)
istruzione;

In C e C++ il ciclo for è molto flessibile. Può semplicemente eseguire le instruzioni all'interno del ciclo al variare della variabile di controllo tra un valore iniziale ed un valore finale oppure può eseguirle fino a che non si verifica una condizione di uscita (NOT(condizione di ciclo)).
Normalmente il valore iniziale ed il valore finale sono variabili intere, inoltre devono soddisfare le condizioni seguenti:

valore iniziale<valore finale
se la variabile di controllo cresce nel ciclo

valore iniziale>valore finale
se la variabile di controllo decresce nel ciclo

E' obbligatorio specificare l'incremento della variabile di controllo.

Esempio

```
/* programma  per for*/

int main (void){
int n,i;
float h;
            scanf("%d",&n);
            printf("%d",n);
            h= 0;
            for (i=1; i<=n;i++)
                  h = h+1/i;  /*oppure h+=1/i */
            printf("Somma=%e \n",h);
            return(1);
}
```

Si può effettuare il ciclo anche per valori decrescenti della variabile di controllo

Esempi

```
for (i=n; i>1;i--)

/*decrementa i di 1 ad ogni passaggio*/


for (i=n; i>1;i=i-3)
/*decrementa i di 3 ad ogni passaggio*/
```

oppure fino al verificarsi di una condizione:

si *esce* dal ciclo appena la condizione di ciclo diventa *falsa*.

Esempi

```
for (i=n; (i>=1)||(n==4);i--)
/*esce se i<1 ed n è diverso da 4*/

for(i=1;(i<100)&&(h<10);i++)
/*esce se i>=100 oppure se h>=10*/
```

Alternative

Anche per le alternative ci sono varie istruzioni che vengono usate per scopi diversi :

```
if (condizione ) istruzione;
                          else istruzione;
```

oppure, in una versione leggermente semplificata,

```
switch (espressione case)
{
case const_1:  istruzione1;
     break;
case const_2:  istruzione2;
     break;
case const_N:  istruzioneN;
     break;
default :  istruzione _alternativa;
   break;
}
```

1. **if** (condizione) istruzione;

   if (condizione) istruzione;

   else istruzione;

L'istruzione viene eseguita *solo se* l'espressione logica è vera, se l'espressione logica è falsa viene eseguita l'istruzione che appare dopo else

L'else e l'istruzione che segue *non* sono obbligatori.

Esempio

```
if (a+b > 10 )
        tronc = 10 ;
    else
        tronc = a+b;
```

Nel caso istruzione sia una istruzione composta va messa tra { }

Esempio

```
if (a+b > 10)  {
            tronc = 10;
            printf("Il troncamento e' pari  a :%f", tronc);
            }
  else
            { tronc = a+b;
              printf("Il troncamento e' pari  a : %f", tronc);
            }
```

Si possono concatenare più istruzioni del tipo

if ... else

Esempio

```
if (10<=a+b  10 &&  a+b=<20)
        x = a+b;
else
        if  (a+b>20)
            x = 20;
        else
            x = 10;
```

In questo modo x=a+b quando $10 \leq a+b \leq 20$, x=20 quando a+b non appartiene all'intervallo [10,20] ed è maggiore di 20, x=10 altrimenti.

2. **switch** (espressione case)

```
{
case const_1:  istruzione1;
     break;
case const_2:  istruzione2;
     break;
case const_N:  istruzioneN;
     break;
default :  istruzione _alternativa;
  break;
}
```

Questo modo di scrivere un'alternativa permette di distinguere molti casi senza dover concatenare troppi if...else

L'espressione case viene valutata per decidere di quale caso si tratti.
Se il suo valore coincide con il valore della costante const i allora vengono eseguite le istruzioni corrispondenti a quel blocco fino al break.

Se il suo valore non coincide con nessuna delle costanti previste, vengono eseguite le istruzioni corrispondenti al blocco default.

Un blocco di istruzioni (istruzione composta) corrispondente ad un case non richiede le {} per segnare l'inizio e la fine del blocco.

Non appena viene eseguito un break il controllo del programma passa all'istruzione seguente. Un blocco case può contenere più di un break.

E' spesso utile scegliere tra molte alternative diverse, come ad esempio nel caso dei menu per la scelta di funzioni e/o nelle opzioni all'interno dei programmi.

L'espressione può essere una variabile numerica oppure una variabile char. E' bene definire all'inizio un nuovo tipo che contenga i possibili valori dell'espressione.

Esempio

```
switch (opzione){
  case 1: f=x*x+2*x+1;
          break;
  case 2: f= sin (x); c=a+b;
          break;
  case 3: f= cos (x);
          break;
default: f=x;
          break;}
```

Quando opzione vale 1 viene posto f=x*x+2*x+1, quando opzione vale 2 viene posto f=sin(x), quando opzione vale 3 viene posto f=cos(x), se opzione vale qualcosa di diverso viene posto f=x.

Ancora sui puntatori

Come abbiamo visto, le variabili possono essere anche di tipo *puntatore ad una variabile*.

Come vedremo l'uso dei puntatori è essenziale per poter modificare il contenuto delle variabili e passarle tra una funzione e l'altra.
In C++ è possibile passare matrici e vettori da una funzione all'altra senza usare i puntatori. Per le variabili scalari invece occorre usare i puntatori.

Ricordiamo che:

- una variabile di tipo puntatore contiene l'indirizzo di memoria di un'altra variabile di tipo int, char, float,...

- non è possibile (nè avrebbe senso) definire una variabile puntatore "universale", cioè che "punti" indifferentemente ad una variabile intera, reale, character,... poiché le occupazioni in memoria di questi tipi sono diverse.
Di conseguenza, a fianco del puntatore bisogna sempre indicare qual è il tipo di variabile cui "punta".

- per indicare che una variabile è una variabile puntatore, nella sua dichiarazione *occorre mettere * prima del nome*.

- l'operatore & applicato ad una variabile restituisce il suo indirizzo di memoria (puntatore alla variabile).

- l'operatore * applicato ad un indirizzo di memoria (contenuto in una variabile di tipo puntatore) restituisce il valore contenuto in quell'indirizzo.

Esempio

```
int *n,*m;
float *x, y, z;
```

definiscono rispettivamente 2 puntatori ad interi (n,m), 1 puntatore ad un reale (x) e 2 variabili reali y,z.

Esempio

Se a è una variabile e pa corrisponde al suo puntatore

&a sarà uguale al contenuto di pa (= indirizzo di a)

*pa sarà uguale al contenuto di a (= valore di a)

Funzioni per la gestione della memoria

In C è essenziale avere un controllo diretto sugli indirizzi di memoria corrispondenti alle variabili.
Due funzioni (della libreria stdlib.h) sono particolarmente utili a questo scopo:

```
sizeof(tipo_della_variabile)
malloc(numero_di_bytes)
```

La funzione sizeof restituisce il valore in bytes dell'occupazione di memoria del tipo di variabile specificata nell'argomento.

Esempio

Il programma seguente manda a video l'occupazione di memoria dei tipi predefiniti in C.

```
#include <stdlib.h>
#include<stdio.h>

int main(void) {
printf("Char: %3d byte\n", sizeof(char));
printf("Integer: %3d byte\n", sizeof(int));
printf("Floating point: %3d byte\n", sizeof(float));
printf("Double precision: %3d byte\n", sizeof(double));
return(1);
}
```

In uscita da main viene assegnato 1 (return(1)) al valore della funzione. E' un modo classico di controllare che sia stato eseguito.

La funzione malloc serve a definire l'allocazione di memoria dei tipi di variabili definiti all'interno del programma.

Di solito viene usata così:

```
malloc(sizeof(tipo_di_variabile))
```

cioè prima viene calcolata l'occupazione di memoria della variabile attraverso la funzione sizeof e poi viene allocata la memoria indicata per quel tipo di variabile dalla funzione malloc.

Attraverso questa funzione è possibile allocare la memoria in modo dinamico (e dunque risparmiare spazio in memoria).

Vettori/Matrici (array)

In C le matrici e i vettori si definiscono indicando il tipo dei loro elementi e le loro dimensioni tra [ ].
Non si possono definire matrici di matrici.

Esempio

```
int vettore_V[10]

int matrice_A [2][10]

float matrice_M [5] [20]
```

sono rispettivamente un vettore a 10 componenti intere, una matrice di interi 2x10 ed una matrice di reali 5x20.

In C il primo elemento di un vettore/matrice ha indice 0, dunque il primo elemento di V è V[0] e l'ultimo è V[9] mentre il il primo elemento di M è M[0][0] e l'ultimo è M[4][19].

L'occupazione di memoria di una matrice nxm è pari a

n x m x occupazione di un reale (o intero)

In realtà, le matrici vengono memorizzate per riga sotto forma di vettori. Come vedremo, questa particolarità rende un po' più complessa una efficiente gestione di matrici/vettori.

Esempio

	1 0 2 3
Matrice A	6 -1 5 4
	7 8 -3 9

diventa in memoria

Vettore a	1 0 2 3 6 -1 5 4 7 8 -3 9

In questo modo l'elemento A[i][j] di una matrice nxm viene memorizzato nell'elemento a[i*m+j] del vettore (perchè il primo indice è comunque 0).

Come vedremo, questa equivalenza è importante, per individuare gli elementi di una matrice attraverso i loro indirizzi di memoria.

Gestione di vettori/matrici in memoria

La gestione della memoria che viene occupata da matrici e vettori (array) è un po' più complessa rispetto ad altri linguaggi perché in C si utilizzano in maniera più trasparente gli indirizzi di memoria delle singole componenti di un vettore/matrice.

In C++ la gestione è semplificata e viene usata più spesso la notazione matematica abituale con due indici. In ogni caso è utile sapere come vengono memorizzati vettori e matrici.

Una matrice A[n][m] è una matrice a n righe e m colonne e gli indici dei suoi elementi A[i][j] sono tali che $0 \le i \le n-1$ e $0 \le j \le m-1$.
Una matrice in C (e in C++) viene trattata come un *vettore di vettori (array di array)* e viene memorizzata per righe in un unico vettore a nxm componenti.

L'elemento A[i][j] della matrice viene quindi memorizzato nell'elemento di posto i*m+j. Il suo indirizzo di memoria è &A[i][j] e corrisponde all'indirizzo di memoria

&A[0][0]+i*m+j

cioè nell'indirizzo che si ottiene sommando all'indirizzo di memoria del primo elemento A[0][0] il numero degli elementi che precedono A[i][j].

Questo è vero *qualunque sia il tipo degli elementi della matrice* e dipende dal fatto che le operazioni sui puntatori tengono conto del tipo di dato cui punta la variabile.

Esempio
Se I è una variabile intera, memorizzata a partire dall'indirizzo 1000, e PI è il puntatore ad I (cioè PI=&I) allora PI+1 punterà all'indirizzo di memoria 1002 (e non 1001) perché un intero occupa 2 bytes.
In modo analogo, nel caso di un reale, PI+1 punterebbe all'indirizzo 1004 perché un reale occupa 4 bytes.

E' bene ricordare che c'è una strettissima relazione tra array e puntatori. Infatti, nella rappresentazione interna di array e matrici, il compilatore C si riconduce *sempre* ad una notazione che fa uso dei puntatori (anche quando nel programma si fa uso degli indici).

Per questo motivo, l'uso dei puntatori al posto degli indici rende il programma più efficiente.

Ecco alcuni esempi di notazioni equivalenti:

Notazione con indici	**Notazione con puntatori**
int a[10]	int *a; a=malloc(sizeof(int)*10));
a[0]	*a
&a[0]	a
a[7]=15	*(a+7)=15

Quando si vuole passare un array ad una funzione è possibile passare soltanto l'indirizzo di un elemento dell'array, indipendentemente dalla notazione utilizzata.

Ecco un esempio di due programmi equivalenti che utilizzano differenti notazioni. In ambedue i programmi viene prima inserito un vettore e poi vengono stampate tutte le sue componenti.

Esempio

```
/*Programma 1: notazione con puntatori*/
#include <stdlib.h>
#include <stdio.h>

/*dimensione massima del vettore*/
#define MAX_DIM 100
int leggi(float *x) {   /* x e' un puntatore reale*/
int n,i;

printf("Numero di elementi:");
scanf("%d",&n)
printf("Inserisci %d numeri",n);
for(i=0;i<n;i++)
/*scanf ha come argomento gli indirizzi delle componenti*/
scanf("%f",x+i);
return(n);
}
```

```
void stampa(float *x, int n){
  int i;
  for(i=0;i<n;i++);
            printf("%f", *(x+i));
  printf("\n");
  return;
}
int main(void) {
  int n,i;
  float v[MAX_DIM];
  n=leggi(&v[0]);
  printf("I %d numeri inseriti sono: ", n);
  stampa(&v[0],n);
  return(1);
}
```

Ecco il secondo programma

```
/*Programma 2: notazione matriciale*/
#include <stdlib.h>
#include <stdio.h>

/*dimensione massima del vettore*/
#define MAX_DIM 100

/* notazione matriciale per il vettore reale x*/
int leggi(float x[]) {
int n,i;

printf("Numero di elementi:");
scanf("%d",&n);
printf("Inserisci %d numeri",n);
for(i=0;i<n;i++)

/*notazione per componenti del vettore*/
scanf("%f",&x[i]);
return(n)
}
```

```
void stampa(float y[], int n){
  int i;
  for(i=0;i<n;i++);

/*notazione per componenti del vettore*/
            printf("%f", y[i]);
  printf("\n");
  return;
}
int main(void) {
  int n,i;
  float v[MAX_DIM];
     /*v è una notazione equivalente a &v[0]*/
n=leggi(v);
printf("I %d numeri inseriti sono: ", n);

/*v è una notazione equivalente a &v[0], dunque
viene passato a stampa () l'indirizzo di v[0]*/
  stampa(v,n);
  return(1);
}
```

Passaggio di variabili nelle funzioni

Come abbiamo già visto, un programma C è articolato in sottoprogrammi che vengono chiamati funzioni.

E' naturale passare le variabili da una funzione ad un'altra oppure dal programma principale (che in C è comunque una funzione) ad un'altra funzione. La prima funzione viene solitamente indicata come *funzione chiamante* .

In C il passaggio delle variabili è per valore, per questo motivo eventuali modifiche delle variabili passate potrebbero rimanere inaccessibili alla funzione chiamante.
Se si vuole permettere ad una funzione di cambiare una variabile `x` occorre passarne l'indirizzo in memoria, cioè il programma chiamante deve passare alla funzione `&x` (e non `x`).

Esempio: Passaggio per valore

```
float media(float x,y,z){
        return((x+y+z)/3)
        }
```

la chiamata a questa funzione è

```
media(a,b,c)
```

dove a,b,c, devono essere tre reali in semplice precisione (float). Il comando return serve ad assegnare a media il valore tra () all'uscita dalla funzione.

In questo modo vengono create tre variabili di appoggio temporanee per contenere x,y,z. All'uscita della procedura quelle variabili vengono cancellate e non sono più recuperabili. Dalla procedura esce solo il valore della media.

Passaggio per indirizzo

Viene usato quando si vuole recuperare il contenuto delle variabili che vengono passate alla fine della esecuzione della funzione. Questo permette di utilizzare i valori in uscita in altre funzioni.

Esempio

```
lettura_dati(float *x,*y){ /* x e y sono puntatori  reali*/

printf("inserisci il valore della prima variabile \n");
scanf("%e",x);
   printf("inserisci il valore della seconda variabile
\n");
   scanf("%e",y);
return()
}
```

la chiamata alla funzione è

```
lettura_dati(&a,&b)
/*vengono passati gli indirizzi*/
```

ovviamente a, b devono essere variabili reali (float) altrimenti il tipo non corrisponderebbe alla definizione della funzione.

In questo modo le variabili non vengono create perché già esistono (come indirizzi) e sono utilizzabili anche in seguito. Si evita la duplicazione.

In C++ matrici e vettori vengono automaticamente passate per indirizzo di memoria (senza che si debba specificarlo).

Esempio

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define MAXDIM 100

typedef float vettore [MAXDIM];
typedef float  matrice[MAXDIM][MAXDIM];

int leggidim(int);
void leggimatrice(matriceA,int n);
float ricercamassimo (vettore_v,int n);

int main {....}
void leggimatrice(matriceA,int n)
{...}
```

Si può passare ad una funzione un numero arbitrario di variabili, alcune saranno passate per valore altre per indirizzo.

Esempio

La chiamata

```
funzione_A(3,&x)
```

alla funzione

```
int funzione_A(int n;float *f)
```

è (almeno formalmente) corretta perché passa un valore ed un indirizzo.

Ricorsività

Si parla di funzione ricorsiva quando *una funzione richiama se stessa*.

Supponiamo che la funzione p che vogliamo definire sia l'elevamento a potenza, x^n.

Ovviamente possiamo scrivere una assegnazione del tipo

$$p = x*x*x*x*x.....*x$$

n volte

In modo più elegante, possiamo definire x^n in *modo ricorsivo* come

$$x^n = \begin{cases} 1 \text{ se } n=0 & \text{per ogni } x \\ x * x^{n-1} & \text{altrimenti} \end{cases}$$

Il linguaggio *C*, come altri linguaggi moderni, permette di scrivere funzioni ricorsive.

Vediamo come possiamo definire una funzione ricorsiva per x^n

Esempio

Supponiamo che n sia >=0. La funzione

```
float  p(float x; int n) {
        float app;
           if (n=0)  app=1;
                    else
                            {app=x;
                            for(i=1;i<n;i++)
                            app= x*app }
        return(app);
}
```

richiede una variabile di appoggio (app).

La sua versione ricorsiva

```
float  p(float x; int n) {   /*indirizzo 1*/
        if (n>0)
                return(x*p(x,n-1));  /*indirizzo 2*/
        else
                return(1);
}
```

non la richiede ed è molto più compatta. Si osservi anche che ambedue le funzioni producono un risultato sbagliato per n<0.
La chiamata alla funzione è, ad esempio, p(y,4).

Bisogna sapere che la definizione di una funzione ricorsiva è certamente più elegante ma *non necessariamente* più efficiente dal punto di vista del calcolo.

Infatti, scrivere una funzione ricorsiva corrisponde a fare il calcolo in due tempi:

1. Prima si costruisce una tabella (*pila*) in cui c'è l'indirizzo dell'istruzione da cui viene fatta la chiamata (indirizzo 1) e l'indirizzo della istruzione chiamata nella definizione (indirizzo 2).

Ad esempio, per n=4, si ottiene schematicamente la pila

Indirizzo	x	n	p	
indirizzo1	y^4	4		
indirizzo2	y	3	p	$(=y^4)$
indirizzo2	y	2	p	$(=y^3)$
indirizzo2	y	1	p	$(=y^2)$
indirizzo2	y	0	p	(=y)

si scende quindi dal dal livello 4 (y^4) fino al livello 1 (y).

2. Poi si eseguono tutte le istruzioni, dal basso verso l'alto, moltiplicando il risultato del livello precedente (in indirizzo 2) per y.

II valori ai vari livelli della pila sono indicati nell'ultima colonna a destra (y, y^2, y^3, y^4). Il risultato finale viene messo nell'indirizzo 1.

Attenzione!

Se il numero delle iterazioni è noto dall'inizio (come ad esempio nel prodotto matrice per vettore) converrà usare un ciclo for *evitando in questo modo la creazione della pila.*

Appendice: software public domain

Per imparare bene il C/C++ occorre programmare ed imparare a trattare vari tipi di dati.

E' quindi importante avere un compilatore C/C++ e possibilmente una libreria grafica.

Questi sono alcuni software public domain (gratuiti) che potrete trovare in rete

Dev-C++	compilatore C++
GNUplot	libreria grafica

potete scaricare questi software dal centro di calcolo all'indirizzo

www.mat.uniroma1.it/centro-calcolo/

oppure da altri siti (cercate le parole chiave con Google, www.google.com).

Le istruzioni per l'installazione sul vostro PC sono solitamente contenute nel file read.me o install.txt (contenuti nella lista dei file che avete scaricato).

Bibliografia

Queste dispense illustrano alcune delle istruzioni del linguaggio C e costituiscono una veloce introduzione all'uso del linguaggio. Per imparare bene occorre programmare.

Per gli approfondimenti (certamente necessari) si possono consultare i testi seguenti:

L.J. Aguilar, *Fondamenti di programmazione in C++*, McGraw Hill, 2008

A. Domenici, G. Frosini, *Introduzione alla programmazione ed elementii di strutture dati con il linguaggio C++*, Franco Angeli, 2001

J. Hubbard, *Programmare in C++*, Schaum's, McGraw Hill, 2001

H. Schildt, *Linguaggio C - La guida completa*, McGraw-Hill, 2003

M. Liverani, *Programmare in C*, Esculapio Editore, Bologna, 2001

A. Bellini, A. Guidi, *Linguaggio C - Guida alla programmazione*, McGraw Hill, 2006

Per migliorare queste dispense siete pregati di segnalare errori ed omissioni a:

falcone@mat.uniroma1.it

Il primo programma

```
/* Il mio primo programma C*/
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int  main ()
{
   int i,n,p; char a;
   printf ("\n\n\n\n  Quanti numeri pari vuoi vedere ?");
   scanf ("%d", &n);
   printf ("I numeri pari richiesti sono  %d",n);

   for (i=1;i<=n;i++)
   {
      printf{"\n");
      p=2*i;
      printf ("%d", p);
      printf ("\n\n\n");
   }
   scanf ("%c", &a);

   printf ("I numeri dispari richiesti sono  %d",n);

   for (i=1;i<=n;i++)
   {
      printf("\n");
      p=2*i-1;
      printf ("%d", p);
      printf ("\n\n\n");
   }

   system("pause");
   return (1);
}
```

Indice analitico

Note e commenti