Capitolo 5: Sintassi del linguaggio C

5.1 Introduzione

L' operazione di apprendimento di un particolare linguaggio di programmazione deve in qualche misura essere smitizzata. La maggior parte dei linguaggi evoluti, al di là degli aspetti estetici o semantici, sono tra loro molto simili nella struttura logica: gli strumenti che forniscono al programmatore sono spesso del tutto analoghi. La realizzazione di un programma può essere divisa in due parti: la prima consiste nel passaggio da una descrizione formale di un problema ad una descrizione di tipo algoritmico della sua soluzione; tale descrizione algoritmica, dovendo essere eseguita da una macchina "stupida", deve prevedere tutte le possibili variazioni del problema e tutti i possibili errori che si possono incontrare nel giungere alla soluzione, e trattarli adeguatamente. La descrizione algoritmica utilizzerà dei "building blocks" fondamentali (ad esempio il ciclo DO-WHILE) che sono propri del linguaggio di programmazione prescelto. Ora, proprio la similitudine nella struttura logica dei linguaggi permette di affrontare questa prima fase della programmazione in modo del tutto indipendente da quella che sarà la scelta finale del linguaggio. La seconda fase della programmazione, ovvero la realizzazione del codice sorgente scritto in un determinato linguaggio, sarà in qualche modo una pura opera di traduzione dal linguaggio astratto dei "building blocks" al linguaggio reale che si è scelto. Servirà quindi una sorta di vocabolario che, data una operazione fondamentale, ne illustri l' utilizzo e spieghi come realizzarla in un dato linguaggio: questo è lo scopo di queste pagine.

Ciò premesso sembrerebbe legittimo affermare che, noto il linguaggio astratto e dato un vocabolario, si possa programmare in qualunque linguaggio. In realtà non tutti i linguaggi di programmazione sono tra loro strutturalmente uguali: alcuni (ad esempio il BASIC) hanno funzionalità inferiori rispetto al C, altri (ad esempio il C++ e tutti i linguaggi "Object Oriented") posseggono una struttura differente (più flessibile ma più difficile da usare). Vi è tuttavia una famiglia piuttosto ampia di linguaggi che sono del tutto similari al C (tra questi i più diffusi sono probabilmente il FORTRAN e il PASCAL), e che potrete legittimamente affermare di "conoscere" non appena abbiate compreso a fondo la struttura logica comune.

Da queste poche considerazioni si comprende comunque come tutte le disquisizioni sulla superiorità di questo o quel linguaggio siano, nella sostanza, del tutto inutili. La vera scelta sta nel modo di realizzare l' algoritmo che risolve un dato problema, e questa è la vera essenza (e la vera difficoltà) della programmazione. La scelta finale del linguaggio dipenderà da considerazioni di tipo estetico, affettivo, economico o pratico; comunque sarà piuttosto irrilevante. Saper programmare significa essere capaci a risolvere problemi in modo algoritmico (ovvero in modo razionale, controllato e riproducibile, esaminando tutte le possibilità e prevedendo tutte le eccezioni). È un esercizio di logica più che di informatica, e può essere molto utile anche quando non ci si trova davanti ad una tastiera.

Prima di proseguire con l' illustrazione del linguaggio è bene osservare che le note che seguono non pretendono di introdurre tutti gli aspetti e le potenzialità del C, ma solo un sottoinsieme sufficiente ad affrontare le problematiche tipiche delle esercitazioni di laboratorio o della maggior parte delle tesi di laurea. Molti strumenti avanzati non verranno neppure citati, ed in alcuni casi si riporteranno solo sintassi semplificate di alcuni comandi. Sebbene, come detto, ciò che segue dovrebbe risultare sufficiente per la maggior parte delle applicazioni, ulteriori dettagli si possono trovare sul seguente testo, disponibile in biblioteca studenti, come pure su qualunque manuale del linguaggio C.

`A. Kelley, I. Pohl - A BOOK ON C - The Benjamin Cummings Publishing Co. Inc.`

5.2 Il comando per la compilazione

Il comando da usare per la compilazione è **cc**. Questo comando possiede un numero piuttosto grande di varianti che sono illustrate nei manuali o nelle "man pages" (vedi Par. 3.12); qui ci limiteremo alle opzioni fondamentali.

I comandi di compilazione usano l' estensione dei files forniti come input per decidere che tipo di operazioni eseguire; in particolare:

Estensione	Contenuto del file	Operazione svolta dal compilatore
.c	Sorgente C	Pre-compilazione, compilazione e link
.o	Codice oggetto	Link

In generale quindi le operazioni di compilazione e link verranno svolte da un comando del tipo:

```
> cc -o <nome_eseguibile> <sorgente_1> ... <sorgente_n> <oggetto_1> ...
<oggetto_n>
```

L' opzione `-o` specifica il nome del file di output, che nel caso precedente è un eseguibile. In alcuni casi è necessario aggiungere all' eseguibile gruppi di routines prelevati da una libreria (le librerie sono di solito fornite con il sistema operativo o con prodotti software opzionali). In questo caso si utilizza l' opzione `-l`, ed il comando generico diventa

```
> cc -o <nome_eseguibile> <sorgente_1> ... <oggetto_1> ... -l<nome_lib_1> ...
```

Per convenzione, i nomi delle librerie cominciano con `lib` e hanno estensione `.a` o `.so`. Il prefisso `lib` deve essere omesso nella linea di comandi. Le librerie CERN possono essere aggiunte semplicemente usando il comando `cernlib`

```
> cc -o nome_eseguibile sorgente_1 ... oggetto_1 ... `cernlib graflib`
```

È possibile eseguire solo la compilazione, senza effettuare il link, ottenendo così un codice oggetto che potrà essere utilizzato in un secondo tempo per fabbricare un eseguibile; per farlo si utilizza l' opzione `-c`:

```
> cc -c -o <nome_oggetto> <sorgente_1> ... <sorgente_n>
```

Come detto in precedenza, i codici oggetto hanno usualmente estensione `.o`.

Un' altra opzione importante è `-g` che deve essere specificata tutte le volte che si intende utilizzare un debugger simbolico (vedi Par. 3.9) per analizzare il comportamento del programma. Questa opzione dice al compilatore di aggiungere all' eseguibile una tabella di corrispondenza tra i nomi delle variabili e le corrispondenti locazioni in memoria. In questo modo il debugger simbolico vi permetterà di ispezionare e modificare il contenuto di una variabile specificandone il nome. L' opzione `-g` disabilita anche tutte le forme di ottimizzazione; questo fa sìche il programma si comporti esattamente come specificato nel sorgente: risulterà quindi più facile da debuggare, anche se sarà decisamente più lento in fase di esecuzione.

5.3 Formato del file sorgente

Il file sorgente è quello che contiene il codice che deve essere tradotto dal compilatore. Questo file deve essere scritto seguendo alcune regole che dipendono dal linguaggio.

Per il C le regole sono piuttosto blande: i comandi possono essere scritti in qualunque posizione, uno o più per riga; si deve porre un punto e virgola (;) alla fine di ciascun comando. I comandi sono usualmente composti da più elementi, ed uno o più di questi elementi possono essere a loro volta comandi completi; è il caso del comando `if`, che è fatto così :

```
if (espressione-logica) comando;
```

dove `comando` è un qualunque comando C (incluso un altro `if` che viene eseguito se l' espressione logica è vera. Gruppi di comandi possono essere realizzati utilizzando coppie di parentesi graffe ({}); sintatticamente un gruppo di comandi equivale ad un comando singolo; nel caso dell' `if` scriveremo:

```
if (espressione-logica) {
  comando-1;
  comando-2;
  ...
}
```

Da notare che se un blocco è posto alla fine di un comando si può omettere il punto e virgola finale.

Non esistono regole di ordinamento delle istruzioni, a parte il fatto che ogni simbolo deve essere definito prima di essere utilizzato. Le lettere maiuscole e quelle minuscole sono considerate diverse: questo è rilevante sia per i comandi, sia per i nomi delle variabili. È possibile inserire commenti nel programma racchiudendoli tra coppie /* - */. Ad esempio:

```
/* Questo e' un commento */
```

5.4 Precompilazione

L' operazione di precompilazione consiste in una trasformazione del codice sorgente che viene eseguita da un programma specializzato o dallo stesso compilatore prima che la fase di compilazione abbia inizio. Il principale scopo di questa operazione è quello di consentire al programmatore di risparmiare tempo e fatica evitando di scrivere per esteso blocchi identici che si ripetono più volte nel programma. Un modo di ottenere questo risultato consiste nell' utilizzo degli "include files". Si tratta di normali files che contengono i blocchi che devono essere ripetuti. Nel file sorgente vero e proprio si sostituirà a questi blocchi una riga che contiene il nome dell' include file:

```
#include "nome-file"
```

In generale gli include files si troveranno nella stessa directory in cui si trova il file sorgente. In caso contrario il nome del file dovrà contenere la specificazione della directory.

In alcuni casi si utilizzano degli include files che definiscono l' interfaccia tra il linguaggio ed il sistema operativo (per esempio le funzioni di input-output). Questi files hanno nomi standard e si trovano in particolari directories note ai compilatori (e di solito non note agli utenti). Per questi files esiste una sintassi speciale:

```
#include <nome-file>
```

Ad esempio:

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

Il precompilatore del C consente inoltre di definire simboli che possono essere usati all' interno del codice:

```
#define EPSILON 1.0e-5
...
if (x < EPSILON) {
...
```

Tali simboli possono anche servire per decidere se compilare o meno alcune parti del programma:

```
#define DEBUG /* Commentare questa linea se non si
                 vogliono stampe di debug */
...
#ifdef DEBUG
  printf("Errore nella routine xx\n");
#else
  /* Qui si deve fare qualcosa per gestire l' errore */
#endif
```

A seconda che il simbolo DEBUG sia definito o meno, il precompilatore deciderà quali linee inserire nel programma che verrà compilato. In questo modo, cambiando una sola linea del codice, si possono ottenere funzionalità diverse del programma.

Inoltre il comando **cc** consente di attivare un simbolo come DEBUG nell' esempio precedente (ovvero di fare l' operazione corrispondente a `#define DEBUG`) senza neppure modificare il sorgente, usando l' opzione **-D**:

```
> cc -DDEBUG -o test test.c
```

5.5 Struttura del programma

Ogni programma è formato da un insieme di routines (vedi paragrafo successivo); le routines possono a loro volta essere raggruppate in moduli. Un modulo contiene usualmente routines tra loro affini (per esempio che lavorano sugli stessi dati o che svolgono funzioni tra loro correlate); in generale le routines appartenenti ad un modulo condividono un set di variabili globali (vedi Par. 5.10).

In C le routines contenute in uno stesso file sorgente (ovvero in uno stesso file .c) formano un unico modulo. Tutte le variabili dichiarate in un file al di fuori di ogni routine sono considerate globali e possono essere usate da tutte le routines del modulo.

Gruppi di routines o moduli che compongono uno stesso programma non devono necessariamente essere compilati insieme: è possibile compilare ogni singolo file sorgente indipendentemente e poi unire il tutto in un unico eseguibile (vedi Par. 5.2).

Ogni programma completo deve contenere una ed una sola routine, detta "main program", dalla quale partirà l' esecuzione. In C il main program è una routine che si chiama `main`.

```
main() {
  ...
}
```

Il main program (al pari di qualunque altra function) può ricevere dei parametri; in questo caso i parametri vengono specificati nella riga di comandi che si usa per eseguire il programma (che, da un punto di vista logico, potrebbe essere pensata come la function che chiama `main`). Il main program può specificare come argomenti il numero dei parametri specificati nella linea di comando e un vettore di puntatori a `char` che contiene i vari parametri. Il seguente esempio dovrebbe chiarificare il meccanismo: consideriamo il programma `main_args.c`:

```
main(int argc, char **argv) {

  int i;

  for (i=0; i<argc; i++) {
    printf("argv[%d] = %s\n",i,argv[i]);
  }

}
```

lo compiliamo con il comando:

```
> cc -o main_args main_args.c
```

quindi lo eseguiamo con il comando

```
> main_args uno due tre quattro
```

ed otteniamo in output:

```
argv[0] = main_args
argv[1] = uno
argv[2] = due
argv[3] = tre
argv[4] = quattro
```

5.6 Functions

Una function è una porzione di codice che esegue una particolare funzione. Può essere chiamata più volte durante l' esecuzione del programma, dal main program o da altre functions; al termine della sequenza di istruzioni previste il programma continua dall' istruzione immediatamente successiva a quella che ha generato la chiamata. La struttura generale di una function è:

```
tipo nome(def_var1,...,def_varn) {

  /* Def variabili locali */
  ...
  /* Codice eseguibile */
  ...
  return;
  ...
}
```

L' istruzione `return` provoca il ritorno dell' esecuzione al programma chiamante.

Le variabili definite dentro una function sono locali, ovvero visibili solo all' interno della function medesima. Le variabili indicate tra parentesi dopo il nome sono dette parametri. Non sono vere variabili, ma solo nomi simbolici che possono essere usati all' interno della function (vengono anche dette variabili "dummy"); al momento della chiamata il chiamante specificherà quali variabili vere prenderanno il posto dei simboli specificati nella definizione della. A questo proposito bisogna notare che in C le functions ricevono una copia della variabile originale, per cui l' accesso ai parametri è solo in lettura: ogni modifica fatta a un parametro risulta visibile solo all' interno della function, ma non si riperquote sulla variabile originale del programma chiamante. Nel caso si desideri che una function modofichi il valore di una variabile del programma chiamante, si dovrà passare un puntatore a tale variabile (`&nome_var`): il puntatore verrà passato come copia, ma la copia punterà comunque direttamente alla variabile originale, consentendo quindi sia la lettura che la modifica.

Il C richiede una definizione completa dei parametri tra le parentesi che seguono il nome della routine; ad esempio:

```
void esempio(float x,int y) {
  ...
}
```

Per chiamare una subroutine si usa la sintassi

```
  float a;
  int b;
  ...
  esempio(a,b);
  ...
```

I parametri possono essere vettori; in questo caso non è necessario specificarne la lunghezza nella definizione data nella subroutine, in quanto la lunghezza sarà sempre quella del vettore specificato dalla routine chiamante. Per esempio:

```
void esempio(float a1[],int a2[][]) {
  ...
}
```

Si deve osservare che, visto che in C il nome di un vettore non seguito da parentesi indica il puntatore al vettore, si può passare un vettore ad una subroutine via puntatore semplicemente specificandone il nome.
Anche il nome della function deve essere preceduto da una definizione di tipo, in modo da indicare chiaramente il tipo del valore che viene ritornato al programma chiamate al termine dell' esecuzione. Nel caso non si voglia ritornare alcun parametro si specificherà il tipo `void`; altrimenti si potrà specificare un valore nell' istruzione `return`:

```
float cubo( float x ) {
  return x*x*x;
}
```

Il programma chiamante userà il valore ritornato dalla function in modo analogo al valore di una variabile:

```
extern float cubo();
float val,val_cube;
...
val = 10.0
val_cube = cubo(val);
...
```

La function deve essere definita nel programma chiamante, con l'opzione external, a meno che il programma chiamante non sia nello stesso modulo della function.

Una function emula il comportamento di una normale variabile: può essere usata all' interno di espressioni numeriche o logiche, e può anche essere passata come parametro ad un' altra routine; si deve però ricordare che ogni utilizzo di una function corrisponde alla attivazione di una porzione di codice che esegue un calcolo e, alla fine, restituisce un valore. Questa operazione è del tutto differente dall' utilizzo di una normale variabile, che richiede unicamente la lettura di un valore dalla memoria.

5.7 Interfacce esplicite

Nel caso in cui una routine ne chiami un altra appartenente allo stesso modulo, il compilatore conoscerà a priori tipo e dimensione di tutti i parametri della routine chiamata, e potrà quindi eseguire un test di correttezza dei parametri passati (in C questo è vero solo se la routine chiamata precede la chiamante all' interno del modulo).

Nel caso la routine chiamata non appartenga al modulo sarà necessario, al fine di rendere possibile un simile test, la definizione di una interfaccia esplicita secondo la sintassi seguente:

```
void  ext_sub(float x1, float x2[], int *i3);
float ext_fun(float x, float y);
```

È prassi comune tra i programmatori C di raccogliere le interfacce di tutte le functions di un modulo (ovvero di un file `module.c`) all' interno di un include file avente lo stesso nome (`module.h`). Tale include file è quindi inserito con un comando `#include` in tutti i moduli che devono accedere alle functions ivi definite.

5.8 Operatori

Gli operatori sono simboli che eseguono un operazione sulla o sulle variabili cui vengono applicati. Gli operatori che agiscono su una sola variabile si chiamano unari (u); un tipico operatore unario è il NOT. Gli operatori che agiscono su due variabili, come ad esempio l' operatore di somma, sono detti binari (b).

Gli operatori si possono poi dividere in relazionali (R), logici (L), numerici (N) e logici "bit a bit" o "bitwise" (B). Gli operatori relazionali, come ad esempio l' operatore di uguaglianza, sono usati nei tests e servono a formare espressioni che possono assumere il valore vero o falso; questo in C significa un valore intero che è zero per "falso" e diverso da zero per "vero". Gli operatori logici possono essere utilizzati insieme agli operatori relazionali per creare espressioni logiche complesse, ad esempio:

```
if ( a==b || c>=d ) {
  ...
}
```

Gli operatori numerici sono usati nel calcolo di espressioni aritmetiche, sia intere che reali.

Gli operatori bitwise agiscono sui singoli bit delle variabili intere; ad esempio l'operatore OR bitwise restituisce una variabile intera i cui bit sono ottenuti facendo l' or logico dei bit corrispondenti delle due variabili di ingresso.

La tabella 5.1 riassume i simboli usati per i principali operatori.

Nome	Tipo	Sintassi C
Assegnazione	b	=
Test uguaglianza	bR	==
Test non uguaglianza	bR	!=
Maggiore	bR	>
Minore	bR	<
Maggiore o uguale	bR	>=
Minore o uguale	bR	<=
AND logico	bL	&&
OR logico	bL	||
NOT logico	uL	!
Somma	bN	+
Sottrazione	bN	-
Moltiplicazione	bN	*
Divisione	bN	/
Cambio segno	uN	-
Elevamento a potenza	bN	pow(,)
Resto divisione intera	bN	%
Incremento	uN	++
Decremento	uN	−−
Inclusive OR	bB	|
Exclusive OR	bB	^
AND	bB	&
NOT	uB	~
Shift destro	bB	>>
Shift sinistro	bB	<<

Tabella 5.1: I principali operatori in C

5.9 Costanti

Esistono fondamentalmente due tipi di costanti: le costanti numeriche e le costanti alfanumeriche. A loro volta le costanti numeriche possono essere intere o reali, e le costanti intere possono essere espresse in forma decimale, binaria o esadecimale. Di seguito vediamo alcuni esempi di costanti:

```
"Un Esempio" /* Alfanumerica */
123          /* Intera base 10 */
0xfa4        /* Intera base 16 */
1.4234       /* Reale          */
0.345e-4     /* Reale          */
```

5.10 Dichiarazione delle variabili

Una variabile è caratterizzata dal tipo, dalla dimensione, dal modo di allocazione e dalla visibilità o scope. Il tipo specifica la codifica con cui i dati vengono scritti in memoria; la dimensione indica di quanti elementi singoli di un dato tipo si compone una variabile: si possono fabbricare scalari, vettori o matrici a due o più dimensioni; il modo di allocazione indica come il compilatore riserva la memoria relativa alla variabile; lo scope indica quali parti di un programma possono accedere in lettura e/o scrittura alla variabile.
I tipi di variabili disponibili nel linguaggio C sono riportati in tabella 5.2.

```
char          /* intero con segno, 1 byte */
unsigned char /* intero positivo,  1 byte */
short int     /* intero con segno, 2 bytes */
int           /* intero con segno, 4 bytes */
float         /* variabile reale, 4 bytes */
double        /* Variabile reale, 8 bytes */
```

Tabella 5.2: Tipi predefiniti nel linguaggio C

Con i tipi predefiniti si possono costruire scalari, vettori oppure matrici; la forma generica dell' istruzione di definizione di una variabile è:

```
tipo nome[dim][...];
```

Ad esempio:

```
short int k,f[2][3];
float     r[4];
```

Si deve ricordare che in C la numerazione degli indici parte sempre da 0.
In generale, le variabili definite all' interno di una routine sono considerate locali e vengono allocate in modo automatico. Per ottenere una allocazione statica di una variabile locale si usa la sintassi

```
static int w;
```

Le variabili definite dentro a un modulo ma fuori da una routine sono considerate globali e sono visibili in tutte le routines del modulo che seguono la definizione. Le variabili globali sono allocate in modo statico. Possono essere inizializzate utilizzando la sintassi

```
int   npoints = 3;
float x[3] = { 1.0, 2.0, 3.0 };
```

Infine è possibile definire delle costanti usando la sintassi

```
const int      npoints = 4;
const float    pi = 3.14159;
```

È utile ricordare che in C le functions utilizzate e definite in un altro modulo devono necessariamente essere dichiarate. Esempio:

```
extern int fun();
```

5.11 Manipolazione di stringhe

Le stringhe sono variabili contenenti gruppi di caratteri alfanumerici. In in C esiste il tipo carattere, ed una stringa può essere costruita come un vettore di caratteri.

```
char stringa[21];
```

La stringa termina obbligatoriamente con un carattere NULL (il carattere con codice ASCII uguale a 0). La lunghezza specificata nella definizione è quindi la lunghezza massima (anzi, per essere precisi è $1 + L_{MAX}$).
La manipolazione delle stringhe avviene per mezzo di funzioni di libreria che richiedono, all' inizio del file sorgente, la definizione

```
#include <string.h>
```

Le stringhe possono essere inizializzate durante la definizione:

```
char stringa[21] = {"Valore Iniziale"};
```

Osservare come, essendo la costante usata nell' inizializzazione lunga 16 caratteri, soltanto 17 caratteri di `stringa` risulteranno effettivamente inizializzati (16 più il NULL finale) mentre i restanti 4 caratteri non verranno inizializzati.

L'assegnazione di una stringa viene eseguita con i seguenti comandi

```
#include <string.h>
...
char str1[26],str2[26];
...
strcpy(str1,'Nuovo Valore');
strcpy(str2,str1);
```

Sono inoltre possibili alcune operazioni sulle stringhe, che sono illustrate nell' esempio seguente:

```
#include <string.h>

char str1[26],str2[26];
char str3[50];
int ll;
...
/* Concatenazione */
strcpy(str3,strcat(str1,str2));
/* Estrazione caratteri */
strncpy(str2,str3+9,10);
/* Lunghezza */
ll = strlen(str2);
/* Test di uguaglianza */
if (strcmp(str1,str2)==0) {
  ...
}
/* Test di uguaglianza */
if (strncmp(str1+4,"Test",4)==0) {
  ...
}
```

5.12 Tipi composti

Un insieme di variabili di tipo e dimensione diversa può essere raggruppato in un tipo composto. Dopo la sua definizione il tipo composto potrà essere utilizzato, come un tipo predefinito, per definire variabili. La definizione di un tipo composto si esegue con le istruzioni

```
struct <nome> {
  <definizione elemento 1>
  ...
}; /* il ; e' obbligatorio */
```

dove la definizione degli elementi è una normale definizione di una variabile, che però non riserva

alcuno spazio in memoria, ma crea solo un paradigma che potrà essere usato successivamente per allocare effettivamente dello spazio. Ad esempio, possiamo definire un tipo `STUDENTE` e poi utilizzarlo per definire un vettore di 100 elementi:

```
struct studente {
  char nome[30];
  char cognome[30];
  int matricola;
  int data_di_nascita[3];
};  /* il ; e' obbligatorio */
...
struct studente st[100];
```

Avremo a questo punto a disposizione un vettore di 100 elementi, chiamato `st`, i cui elementi sono di tipo `studente`. Ogni singolo elemento del tipo composto può essere utilizzato come una variabile usuale, utilizzando la sintassi

```
st[3].matricola = 132456;
st[10].data_di_nascita[2] = 1972;
```

5.13 Puntatori ed allocazione dinamica della memoria

Un puntatore è una variabile il cui contenuto è l' indirizzo di un' altra variabile.

In C il puntatore è una variabile con un nome proprio, che viene usato solo per indicare il puntatore. La variabile puntata dal puntatore viene indicata con un nome differente, che è dato del nome del puntatore preceduto da un asterisco. Un puntatore C è specifico per un dato tipo (ad esempio `int` o `float`) ma può puntare sia scalari che vettori di quel tipo.

Ecco alcuni esempi di definizione di un puntatore, dove si deve osservare come la definizione di un puntatore ad uno scalare è identica alla definizione di un puntatore a vettore:

```
short int * s_point;
double    * v_point;
```

Vale la pena di osservare che non è possibile fabbricare puntatori a matrici; in realtà un oggetto del tipo

```
float  **matrix;
```

è un puntatore a oggetti di tipo "puntatore a float". `matrix` può quindi essere usato per fabbricare un vettore di puntatori, e ogni puntatore può a sua volta puntare a un vettore di float. Questo realizza una struttura che ha due indici, come una matrice, ma di fatto è più complessa di una matrice (ad esempio le righe, ovvero i vettori puntati dai vari puntatori, possono essere di lunghezza diversa).

L' assegnazione di un puntatore può avvenire tramite allocazione esplicita di una porzione di memoria; ad esempio le istruzioni

```
float *r_point;
...
r_point = malloc(100*sizeof(float));
```

chiedono al sistema di riservare (al momento dell' esecuzione) una porzione di memoria corrispondente a 100 reali (400 bytes) e di assegnare il puntatore `r_point` in modo che punti a quest' area. Si noti come è necessario passare alla funzione `malloc` il numero di bytes da allocare, il che può essere fatto sia direttamente sia usando la funzione `sizeof` che ritorna la lunghezza in byte di un dato tipo.

Un area allocata in questo modo (dinamicamente) deve essere liberata quando non serve più utilizzando le istruzioni

```
free(r_point);
```

Un puntatore può poi essere assegnato eguagliandolo ad un altro puntatore; ad esempio:

```
int *a, *b;
...
a = b;
```

Dopo l' assegnazione, `a` e `b` puntano alla stessa locazione di memoria; di conseguenza possono essere usati in modo interscambiabile nel resto del programma.

Un terzo modo di assegnare un puntatore è quello di farlo puntare ad una variabile allocata staticamente. Per esempio:

```
int *a;
int b;
...
a = &b;
```

In C il simbolo `&` seguito dal nome di una variabile indica l' indirizzo di quella variabile.

Un quarto modo di assegnare un puntatore è quello di fornire esplicitamente l' indirizzo in memoria della variabile puntata. Questa operazione è utile nel caso si debba manipolare un dispositivo mappato in memoria in una posizione fissa. Anche in questo caso il C esegue l' assegnazione del puntatore come quella di una normale variabile:

```
int *register_1;
...
register_1 = 0xff042000;
```

Come già detto, in C ogni riferimento alla variabile puntata viene fatto anteponendo un asterisco al nome del puntatore. Fanno eccezione vettori e matrici, nel senso che se `a` è un puntatore `a[3]` è la quarta componente del vettore puntato da `a`. Questo fatto è vero anche per i vettori allocati staticamente: se `b` è definito come in FORTRAN sia sufficiente specificare quante locazioni di un dato tipo (`REAL` nell' esempio) si desiderano, ed il computo dei bytes viene fatto automaticamente. In C invece

```
int b[100];
```

allora il simbolo `b` senza parentesi indica il puntatore al vettore, ovvero le due scritture

```
b e &(b[0])
```

sono equivalenti.

Vediamo ora alcuni esempi di uso dei puntatori:

```
int *a,*b;
int c;
...
b = malloc(10*sizeof(int));
a = &(b[4]);
...
b[0] = b[c] + c;
*a = b[2] + b[3];
...
fun(*a);
...
```

Si possono definire puntatori a tipi composti, come pure si possono fabbricare tipi composti che abbiano puntatori come elementi; rifacendoci all' esempio di Par. 5.12 avremo

```
struct studente *pt_s;
...
pt_s = malloc(sizeof(struct studente));
...
strcpy(pt_s->nome,"Luisa");
```

Si noti la sintassi particolare che, nel caso di un puntatore, sostituisce `nome.` con `nome->`.

5.14 Istruzioni condizionate e cicli

Le istruzioni condizionate sono blocchi di istruzioni che vengono eseguiti solo se si verificano particolari condizioni. Le condizioni che devono essere verificate sono date da espressioni logiche (vedi Par. 5.8). La sintassi è:

```
if (espressione_logica) {
  istruzioni
  ...
} else {
  istruzioni
  ...
}
```

I cicli sono gruppi di istruzioni che vengono ripetute più volte; esistono due tipi di cicli: quelli per cui si conosce a priori il numero di ripetizioni (a priori significa prima di cominciare la prima iterazione) e quelli per cui il numero di ripetizioni non è noto a priori. Vediamo le sintassi corrispondenti.

```
int i;
...
for(i=val_iniz; i<=valore_fin; i=i+passo) {
  istruzioni
  ...
}
```

```
do {
  istruzioni
  ...
} while (espressione_logica);

while (espressione_logica) {
  istruzioni
  ...
}
```

Il ciclo `do-while` si ripete fino a che l' espressione logica assume il valore "falso".

5.15 Input/Output su terminale

L' Input/Output (I/O) è l' operazione con la quale si trasferiscono i dati contenuti nella memoria su un terminale o un file e viceversa. Esistono due tipi di I/O: quello formattato e quello non formattato. Nell' I/O formattato un numero viene rappresentato in cifre; ad esempio un numero

reale può essere rappresentato come "0.00012945" (10 caratteri, cioè 10 bytes) o come ".13e-3" (6 bytes) malgrado il fatto che la rappresentazione di un reale in memoria sia sempre lunga 4 bytes. In questo caso naturalmente il programmatore deve decidere i dettagli del formato che vuole utilizzare (ad esempio quante cifre mostrare). Nell' I/O non formattato invece i dati vengono letti o scritti esattamente nello stesso formato usato in memoria (per esempio un reale viene sempre scritto in 4 bytes); ovviamente, nel caso non formattato i dati sono del tutto illeggibili, ma occupano meno spazio; inoltre ciò che viene scritto ha esattamente la stessa precisione di quanto contenuto in memoria, laddove l' output formattato ha usualmente una precisione inferiore. Nel seguito ci occuperemo esclusivamente di I/O formattato.

Il C fornisce istruzioni per leggere variabili dalla tastiera e scriverle sullo schermo del terminale (più precisamente queste istruzioni scrivono e leggono su due canali di comunicazione, detti STDIN ed STDOUT che sono usualmente assegnati alla tastiera e allo schermo). Il formato delle istruzioni è:

```
#include <stdio.h>
...
scanf("formato",&var_1,...,&var_n);

printf("formato",var_1,...,var_n);
```

Notare che è necessario includere il file `stdio.h` in ogni modulo che esegue operazioni di I/O; notare pure che la funzione `scanf` deve ricevere i puntatori alle variabili da leggere (`scanf` è una funzione che deve modificare il valore di alcuni ei suoi parametri).

La specificazione del formato è molto utile in fase di output, in quanto consente di presentare i dati in forma ordinata ed esteticamente piacevole. In input invece specificare un formato significa richiedere che i dati vengano immessi ESATTAMENTE nel modo specificato, e questo può risultare piuttosto scomodo. Per questo motivo in input si usano i cosiddetti "formati liberi", che consentono di scrivere i dati in un modo quasi arbitrario lasciando al computer il compito di fare del proprio meglio per interpretare correttamente quanto viene scritto in ingresso. In C è comunque necessario essere relativamente precisi, utilizzando `%d` per ogni numero intero, `%f` per ogni float, `%lf` per ogni double e `%s` per ogni parola (cioè sequenza di caratteri alfanumerici che non contiene spazi). Inoltre la sequenza di caratteri in input viene letta dopo che viene premuto il tasto "Enter"; è buona norma includere anche questo carattere alla fine di ogni formato di input (lo si fa aggiungendo un `%*c`) in modo che le operazioni successive non ricevano residui degli input precedenti.

Nel seguito vediamo alcuni esempi di utilizzo dei più comuni formati di output. Si deve notare che, oltre ai formati delle variabili da stampare, è possibile includere nella specifica del formato anche caratteri normali che vengono stampati senza alcuna interpretazione. Inoltre in C l' output di successive istruzioni `printf` avviene di seguito sulla stessa riga, a meno che il formato non termini con `\n`.

```
#include <stdio.h>
#include <string.h>
...
int    i1;
float  r1,r2;
char   c1[21];
...
i1 = 123;
r1 = 1.23456789;
r2 = 283.3344;
strcpy(c1,"Prova di I/O");
...
printf("i1 = %5d \n", i1);
printf("r1 = %5.3f \n", r1);
printf("r2 = %12.5e \n", r2);
printf("s1 = %s* \n", c1);
printf("%10.5f%10.5f \n", r1,r2);
```

Il progamma sopra scritto produce il seguente output:

```
i1 =   123
r1 = 1.235
r2 =  2.83330E+02
s1 = Prova di I/O*
  1.23457 283.33441
```

5.16 Input/Output su file

L' I/O su un file è del tutto analogo a quello su terminale. La sola differenza consiste nella necessità di definire il nome del file sul quale si desidera agire; a tale file verrà assegnato un identificatore di canale di comunicazione (di solito si tratta di un numero intero, che in C è considerato essere un puntatore ad un tipo di variabile detta `FILE`) che verrà utilizzato in tutte le operazione relative a quel file. Una volta terminate le operazioni di I/O si dovrà chiudere il canale (ed il file relativo).

Di seguito vediamo come si eseguono le operazioni di apertura e chiusura del file.

Le variabili indicate nell' esempio con i nomi `stat1` e `stat2` indicano codici di errore. Valori negativi indicano che qualcosa è andato storto.

```
#include <stdio.h>
...
FILE *f1,*f2;
int stat1,stat2;
...
f1 = fopen("nome_file_in","r");

f2 = fopen("nome_file_out","w");

...
stat1 = fscanf(f1,"formato", &var_1,...,&var_n);
stat2 = fprintf(f2,"formato", var_1,...,var_n);
...
fclose(f1);
fclose(f2);
```

Nel caso si debba leggere una linea generica di un file, che possa contenere spazi si utilizzeranno le istruzioni:

```
  char line[81];
  ...
  fgets(line,80,f1);
```

Osserviamo infine che è possibile, come variante della scrittura e lettura su file, scrivere su o leggere da una stringa di caratteri, come illustra l' esempio seguente:

```
#include <stdio.h>
...
char s1[21],s2[21];
...
sscanf(s1,"formato", &var_1,...,&var_n);
sprintf(s2,"formato", var_1,...,var_n);
```

Naturalmente è responsabilità del programmatore accertarsi che le stringhe siano lunghe abbastanza per contenere i dati prodotti dai formati di output.