IL LINGUAGGIO C

Il linguaggio C fu progettato e sviluppato da D.Ritchie ai Laboratori Bell, quale linguaggio di programmazione di Unix

Il C è un linguaggio imperativo a blocchi simile al Pascal; presenta caratteristiche linguistiche sia di basso livello che di alto livello

- È particolarmente adatto per scrivere programmi di sistema grazie ai suoi meccanismi di controllo dell'hardware al livello più basso

- Anche se permette al programmatore di sistema un controllo diretto dell'hardware, è in gran parte indipendente dalla macchina

- Ha strutture di controllo di alto livello, mentre il sistema dei tipi è sostanzialmente debole (non ha tipaggio forte)

- Ha uno standard comunemente accettato perché esiste un'unica fonte di definizione (dal 1988 esiste ANSI C)

- È abbastanza ricco da costituire uno strumento serio nello sviluppo di software per applicazioni pratiche

- Mediante il meccanismo delle librerie permette innumerevoli estensioni linguistiche. Per esempio, attualmente si sta diffondendo una evoluzione del C, il linguaggio C++, che include meccanismi per programmazione ad oggetti

- La maggior parte di Unix e di Linux è scritta in C (il resto è in assembler); Unix è fatto in modo da costituire una libreria del C, grazie alle chiamate di sistema

Struttura dei programmi C

Un programma C è costituito da una **collezione di funzioni**
(invece in Pascal: programma = dichiarazioni + blocco)

Il programma da eseguire (blocco principale) è definito dalla funzione denominata **main**

```
main()        /* questo è un commento */
{    int i, j, k;  /* dichiarazione di 3 var. intere */
     i = 10;       /* imposta i a 10 (decimale) */
     j = i + 015;         /* imposta j a i + 15 (ottale) */
     k = j * j + 0xFF;    /* imposta k a j * j + 0xFF */}
                                /*    (esadecimale) */
```

Le funzioni contengono **dichiarazioni**, **istruzioni**, e altri elementi

Le **variabili** devono essere dichiarate prima di essere usate

Tutte le istruzioni devono essere *terminate* da un punto e virgola
(invece in Pascal ";" si usa come *separatore*, non come terminatore)

I **commenti** iniziano col simbolo "/*" e terminano col simbolo "*/";
si possono estendere su più linee

Dichiarazioni

Formato delle dichiarazioni:

```
Tipo Identificatore
```

Esistono 4 **tipi primitivi:**

- char un carattere rappresentato con un byte
- int un intero la cui ampiezza dipende dalla macchina
- float numero in virgola mobile a precisione semplice
- double numero in virgola mobile a precisione doppia

Non esiste alcun tipo primitivo booleano, né il tipo *stringa*
I booleani vengono simulati mediante numeri;
falso si denota con zero; qualsiasi altra cosa sta per il valore *vero*

Le stringhe vanno costruite come vettori di elementi del tipo char
(La libreria a tempo di esecuzione fornisce le operazioni sulle stringhe)

I valori di tipo char si comportano come numeri interi nell'intervallo 0-255 se sono qualificati unsigned, oppure in -128+128 se signed

I valori di tipo char possono essere usati nelle operazioni aritmetico/logiche e possono essere frammiste a numeri interi:
è legale applicare ai dati di tipo char le operazioni aritmetiche, o confrontarli logicamente con altri caratteri o interi

Alcuni tipi primitivi possono ricevere uno o più **qualificatori**

Qualificatore	**Funzione**	**Tipi cui si applica**
`short`	compatta gli interi	`int, float`
`long`	aumenta la precisione	`int, float`
`unsigned`	il bit di segno è usato come dato	

Tipi strutturati

Esistono 4 modi per costruire tipi strutturati:

- puntatori
- vettori
- strutture (in Pascal: record)
- unioni (in Pascal: record varianti)

I **puntatori** sono astrazioni degli indirizzi macchina

Un **vettore** è una sequenza indicizzata di elementi dello stesso tipo
Tutti i vettori iniziano con l'elemento 0

Una **struttura** è una collezione di variabili, di solito di tipo diverso

Un'**unione** è una struttura che può assumere uno tra parecchi tipi in funzione del valore di un campo etichetta (tag)

Puntatori

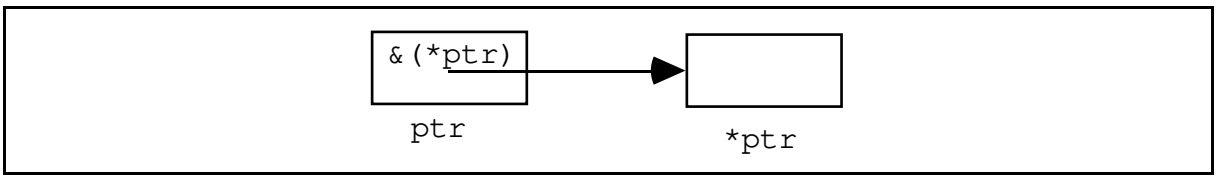
Esempio di dichiarazione di puntatore:

```
char *ptr;
```

Questa dichiarazione dichiara

1) l'oggetto *ptr di tipo char
2) l'oggetto ptr come puntatore

Agli oggetti di tipo puntatore possono essere assegnati valori di tipo indirizzo, denotati dal segno "&".



Esempi:

`int *ip;`	(la variabile ip è un puntatore ad un dato di tipo int)
`int i;`	(la variabile i può contenere dati di tipo i)
`ip = &i;`	(alla variabile ip viene assegnato l'indirizzo di i)
`*ip=17`	(alla variabile i viene assegnato il valore 17)

La forma della dichiarazione di puntatore descrive in sostanza il *cammino d'accesso* al dato dichiarato; se "si esegue mentalmente" un operatore di dereferenziamento (vale a dire un operatore '*') entro una dichiarazione, "si arriva" al dato dichiarato

Analogamente, se "si esegue" la stessa sequenza di operatori di dereferenziamento entro un'istruzione, "si arriva" all'elemento su cui opera l'istruzione stessa

Vettori e puntatori

• In C sono ammessi solamente i vettori monodimensionali
• L'ampiezza di un vettore deve essere fissata a tempo di compilazione
• Si possono costruire vettori di qualsiasi tipo (anche vettori di vettori)

```
int x[30], y[30];
int matrix[30] [30];
```

Se il vettore viene dichiarato come parametro di funzione,
la prima dimensione viene omessa

```
main(x,y,matrix)
int x[], y[], matrix[] [30];
```

• Esistono solo due operazioni primitive sui vettori:
- determinazione della lunghezza;
- accesso al primo elemento (di indice 0).

• *Tutte le altre operazioni sono effettuate mediante puntatori*n (anche se si possono usare normalmente gli indici); in particolare, *non si possono passare* vettori a procedure (ma puntatori sì)

Esempio: Se `a` è il nome di un vettore,

`*a=100;` assegna 100 all'elemento a[0]
`*(a+1)=100;` assegna 100 all'elemento a[1]

Sono equivalenti i seguenti assegnamenti:

```
i = vettore[4][7];
i = *(vettore[4]+7);
i = *(*(vettore+4)+7);
```

Uso dei puntatori

La nozione di puntatore permette

- di usare i puntatori come vettori
- di manipolare i puntatori in modo indipendente dalla macchina.

Qualsiasi puntatore p può essere usato con la notazione

```
p[i]
```

per accedere all'i-esimo elemento nel blocco cui si riferisce p

Esempio di manipolazione di stringhe mediante puntatori:

```
char *p, *q;
p= "xyz";
q=p;
q[1]='Y';  (Adesso sia p che q puntano alla stringa "xYz")
```

Il nome di un vettore si può trattare
o come *vettore*, usando l'operatore [],
oppure come *puntatore* usando l'operatore *

Quindi dereferenziando p mediante l'operatore * si ottiene lo 0-esimo elemento del vettore, cioè p[0]

Operazioni su puntatori

I puntatori vengono aggiornati in modo indipendente dalla macchina

Supponiamo che p sia un puntatore a oggetti di tipo intero (`int`), e che la macchina che stiamo usando rappresenti gli interi su due byte: l'incremento di p risulta in un nuovo puntatore che punta all'intero successivo nel vettore

L'operazione di incremento (scritta `p++`) non aggiunge 1 all'indirizzo rappresentato da p: aggiunge invece all'indirizzo rappresentato da p il numero intero di byte necessari a rappresentare in memoria un'istanza del tipo intero, cioè il valore 2

Ne segue che p[i] e *(p+i) accedono allo stesso dato.

Esempio:

se p contiene il valore 0x40440 prima dell'incremento,
dopo l'incremento p contiene il valore 0x40442.

Espressioni con effetti laterali

Esempio:

```
while (n--) *p++ = *q++;
```

Le variabili *p* e *q* sono puntatori a caratteri, *n* è un contatore
Il ciclo copia una stringa di *n* caratteri dalla locazione puntata da *q* alla locazione puntata da *p*.

Esempio:

```
for (i = 0; i < N; i++) a[i] = 0;
```

Imposta a0 i primi *N* elementi di *a*

Possiamo scrivere questo ciclo anche nel modo seguente

```
for (p = &a[0]; p < &a[N]; p++) *p = 0;
```

Il puntatore *p* viene inizializzato in modo che punti all'elemento 0 del vettore. Il ciclo continua fino a che *p* non ha raggiunto l'indirizzo di *a*[*N*], che è il primo elemento escluso

Nota bene:
A causa della sintetiticità del linguaggio, gli assegnamenti possono apparire in posti inaspettati:

```
if (a = b) istruzione;
```

assegna *b* ad *a* e quindi controlla *a* se è diverso da zero.
Il condizionale di cui sopra è totalmente differente dal seguente:

```
if (a == b) istruzione;
```

che confronta 2 variabili ed esegue l'istruzione se queste sono uguali

Operazioni su puntatori

L'operatore "**.**" (punto) permette l'accesso agli elementi di una struttura

```
impiegato.nome
```

restituisce il valore del campo `nome` della struttura `impiegato`

Se `p` è un puntatore alla struttura `impiegato`, l'espressione

```
p -> nome
```

è equivalente a quella precedente

& restituisce l'indirizzo della locazione cui si riferisce l'operando
In generale valgono le seguenti equivalenze

```
&p[i] = = (p + i) = = &*(p + i).
```

sizeof è una funzione predefinita che calcola l'ampiezza in byte della memoria allocata per un tipo o un dato; permette di scrivere codice indipendente dalla gestione della memoria della macchina sottostante

Esempio:
se `p` è un vettore `sizeof(p)` calcola il numero di byte allocati al vettore

Precedenza degli operatori

Si consideri l'espressione

```
*p[5]
```

Poiché la precedenza dell'operatore [] è maggiore di quella dell'operatore *, verrà calcolato prima p[5] dopodiché verrà applicato l'operatore di dereferenziamento.

Se gli operatori in esame sono dello stesso livello di precedenza, allora la colonna *Associatività* definisce l'ordine in cui vanno eseguiti, cioè o da sinistra a destra (S-D) oppure da destra a sinistra (D-S)

Livello	Tipo	Operatori	Associatività
15	Primario	() [] . ->	S-D
14	Unario	! ~ ++ -- -	D-S
13	Aritmetico	* / %	S-D
12		+ -	S-D
11	Shift	<< >>	S-D
10	Relazionale	< <= > >=	S-D
9		= = !=	S-D
8	Logico su bit	&	S-D
7		^	S-D
6		|	S-D
5	Logico	&&	S-D
4		||	S-D
3	Condizionale	? :	S-D
2	Assegnamento	= += -= *= /= %= ^= &= >>= <<= |=	D-S
1	Virgola	,	S-D

Interpretazione di dichiarazioni complesse

Le regole sintattiche esatte per dichiarazioni complesse che coinvolgono vettori, puntatori e altri tipi sono complicate

```
struct pair
    { int *ssn; char person[30];}
    *(*findary()) [];
```

Si possono definire regole che aiutano nella lettura delle dichiarazioni; l'ordine di applicazione delle regole è determinato dalla precedenza degli operatori

Bisogna cominciare dall'identificatore più interno ed applicare le regole in modo che la parte tradotta si espanda verso l'esterno finché non è stata tradotta tutta la dichiarazione

Forma	**Viene trasformata in**
**rxpr*	<*rxpr* puntatore a>
rxpr[]	<*rxpr* un vettore contenente>
rxpr[iconst]	<*rxpr* un vettore (lungo *iconst*) contenente>
rxpr()	<*rxpr* una funzione che restituisce>
struct *tag rxpr1 rxpr1*	<*rxpr2* struttura definita *tag* con campi *rxpr1*>
struct *tag rxpr*	<*rxpr2* struttura definita *tag*>
struct *rxpr1 rxpr2*	<*rxpr2* struttura con campi *rxpr1*>
union *tag rxpr1 rxpr1*	<*rxpr2* unione definita *tag* con membri *rxpr1*>
union *tag rxpr*	<*rxpr* unione definita da *tag*>
storeclass rxpr	<*rxpr* classe di memoria *storeclass*>
{*rxpr*;}	<*rxpr*>
{*rxpr1*; *rxpr2*	{<*rxpr1* e *rxpr2*>>
(*rxpr*)	<*rxpr*>

Esempio

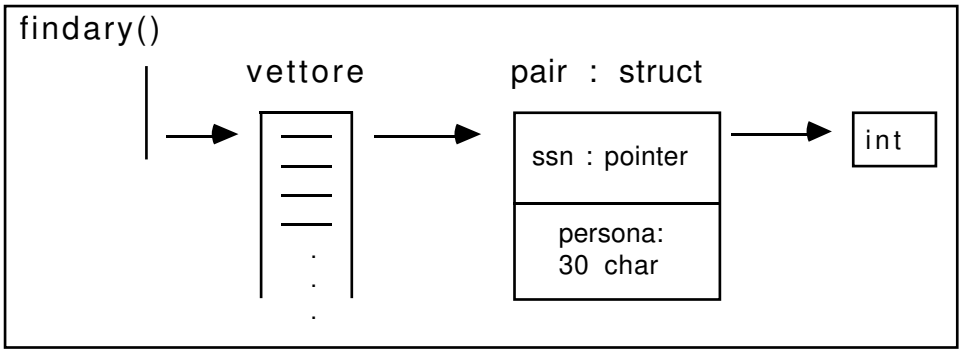
```
struct pair
    { int *ssn; char person[30];}
    *(*findary()) [];
```

L'esempio può essere interpretato operando come segue:

`struct pair { int *ssn; char person[30];}`
*<*findary* è una funzione che restituisce un puntatore ad un vettore …>;

`struct pair`
<*ssn* puntatore al tipo int e *person* vettore (lungo 30) di tipo char>
<*findary* è una funzione che restituisce un puntatore ad un vettore
contenente puntatori a…>;

<*findary* è una funzione che restituisce un puntatore ad un vettore
contenente puntatori a una struttura definita dal tag *pair*
con campi
ssn puntatore al tipo int
e
person vettore (lungo 30) di tipo char>;



Gestione dell'allocazione di memoria

Nelle dichiarazioni delle variabili possono essere usati alcuni qualificatori che hanno effetto sulla gestione della memoria per la variabile che si sta dichiarando

• Il qualificatore `auto` permette l'allocazione automatica di memoria
La memoria si alloca al momento di entrare nel blocco in cui è dichiarata la variabile e si dealloca al momento di uscire dal blocco

• Il qualificatore `static` assegna memoria che non viene allocata dinamicamente
La memoria statica viene allocata all'inizio del programma e sopravvive "staticamente" per tutta l'esecuzione

• Il qualificatore `register` suggerisce al compilatore che il dato associato alla variabile, se possibile, dovrebbe essere tenuto in un registro macchina

• Il qualificatore `extern` permette di usare in una certa unità di programma, compilata separatamente, variabili dichiarate in un'altra unità di compilazione

Se la dichiarazione della variabile non è contrassegnata da nessuna di queste annotazioni, la modalità default è auto

Conversione di tipo (casting)

Si consideri l'espressione

```
(int ) a
```

dove a è un oggetto di tipo qualsiasi. Il risultato è un valore di tipo puntatore a intero, ottenuto a partire dalla rappresentazione di macchina del valore di `a`

L'operatore di **conversione forzata di tipo** permette di esprimere la coercizione della rappresentazione di un oggetto

La conversione può anche essere implicita

Esempio:

```
fraction = x - (float) ((int) x)
```

calcola la parte frazionaria della variabile reale x sottraendo la parte intera di x espressa come numero reale; la parte intera viene prodotta convertendo x in int e poi di nuovo in float

Esempio:

```
int minuscola(c)
int c;
{
 if (c>='A' && c <= 'Z')
        return(c+'a'-'A')
    else return(c)
}
```

cambia un carattere da maiuscolo a minuscolo, passando per la sua rappresentazione in forma `int`

Istruzioni e blocchi

Istruzione semplice

Una dichiarazione, un'assegnazione oppure una chiamata di funzione

Può essere posta ovunque all'interno di una riga oppure su più righe

Termina sempre con ";" (in Pascal ";" è un separatore)

Blocchi

Un blocco è una serie di istruzioni legali racchiusa tra parentesi graffe

{*istruzione1 istruzione2 . . . istruzionen*}

Esempio:

```
{
    xtrows = col / maxcolp1;
    col %= maxcolp1;
    row += xtrows;
    row %= maxrowp1;
}
```

L'istruzione if

```
if (expr)              /* if semplice */
    istruzione

if (expr)              /* if generale */
        istruzione1
    else
        istruzione2
```

L'espressione condizionale dopo l'**if** va racchiusa tra parentesi tonde

La prima istruzione dell'**if** segue immediatamente dopo l'espressione condizionale senza parole chiave di riferimento; non si usa **then**

Si possono usare le parentesi per eliminare l'ambiguità sull'associazione delle parti **else** con le parti **if**

Senza le parentesi, **else** va associato all'**if** più vicino

```
if (col = = incol)     /* if semplice */
        {putchar(c); return;}

if (c = = CTRLZ)       /* if generale */
        return(EOF);
    else
        return(c);

if (isalpha(c))              /* else-if */
        return(LETTER);
    else    if (isdigit(c))
            return(DIGIT);
        else
            return(c);
```

L'istruzione switch

```
switch (expr)
    {
    case const1:   istruzione1
    case const2:   istruzione2
                   .
                   .
                   .
    default: istruzione-n
    }
```

Per uscire da un'istruzione **switch** si usa un'istruzione **break**

```
switch (iir)
    {
    case '\006':
             note_error(iir);
             break;
    case '\004':
             buffer_data();
             break;
    case '\002':
             thrfree = 1;
             break;
    case '\000':
             msr_change();
             break;
    default: return;
    }
```

NOTA BENE: Se l'ultima istruzione di un blocco non è un trasferimento di controllo (cioè **break** o **return**), il controllo passa all'insieme di istruzioni successivo

Ciclo while

L'istruzione `while` ha due forme
(la seconda delle quali è simile all'istruzione `repeat` del Pascal)

While semplice

```
while (expr) istruzione
```

```
int argc;
char *argv[];
while (--argc > 0 && (*++argv)[0] == '-')
    {   /* Opzioni di elaborazione */
        … }
```

Questo ciclo **while** continua ad eseguire il corpo fintantoché

il numero degli argomenti da elaborare (cioè **- -argc**) è maggiore di zero e il primo carattere della stringa corrente (cioè **(*++argv)[0]**) è un "meno"

Ciclo do-while

```
do
    istruzione
while (expr);
```

Il ciclo **do-while** esegue l'*istruzione* almeno una volta
La condizione viene controllata alla fine del ciclo

L'esempio che segue termina una stringa che viene scritta su **stdout** e la fa seguire da **i** caratteri spazio, con **i** maggiore o uguale a zero

```
do
    putchar('\n');
while (i -- > 0);
```

Ciclo for

L'istruzione for è diversa da quella del Pascal

```
for (expr_iniziale ; expr_while; expr_aggiornamento)
            istruzione
```

L'espressione *expr_iniziale* viene eseguita prima del ciclo,
l'espressione *expr_while* viene eseguita all'inizio di ciascun ciclo,
l'espressione *expr_aggiornamento* viene eseguita alla fine di ogni ciclo
Il ciclo continua fintanto che il valore di *expr_while* è diverso da zero

Esempi:

```
for (i = 0; i < n; i = i + 1)
    a[i] = 0;

for (;;) {comandi}          /* ciclo infinito */

for (p=buffer+row; p<bufptr; p+= NROWS) print(*p);

for (s = argv[0]+1 ; *s != '\0' ; s++)
    print(*s);
```

Istruzioni break e continue

L'istruzione **break** forza l'uscita dal più interno costrutto di tipo **for**, **while**, **do-while** o **switch;** il controllo viene trasferito subito all'esterno del costrutto interessato
L'istruzione **continue** forza l'inizio immediato dell'iterazione successiva del ciclo più vicino di tipo **for**, **while** o **do-while**

```
# define TRUE 1
# define ENDOP '\117' /* Secondo char del tasto END */
# define CURUP '\110' /* Secondo char del tasto UP */
# define CURLFT '\113'/* Secondo char del tasto LEFT*/
# define CURRGT '\115'/* Secondo char del tasto RIGHT*/
# define CURDWN '\120'/* Secondo char del tasto DOWN*/

char c, getc_noecho();int x, y;        /* Posizione del
cursore (riga, col) */printxy("Sposta il cursore nella
posizione desiderata,            premi END per
terminare", 24, 0);while (TRUE) {      /* L'utente
sposta il cursore */  if ((c = getc_noecho()) != '\0')
            /* Non tasto funzione */
    {send_bell(); continue;}     if ((c =
getc_noecho()) == ENDOP) break;                /* Uscita
dal ciclo */    switch(c) {      case CURUP:       if
(y>1) move_cursor(--y, x);             break;       case
CURDOWN:              if (y<maxrow) move_cursor(++y,x);
            break;          case CURRGT:            if
(x<maxcol) move_cursor(y,++x);
            break;          case CURLFT:            if
(x>1) move_cursor(y, --x);
            break;          default: send_bell();
    } }
```

Funzioni e programmi

```
tipo func (lista_argomenti)
 dichiarazioni lista_argomenti
 {
 dichiarazioni variabili locali;
 istruzioni
 }
```

func ha sempre un *tipo*, e restituisce un valore di quel *tipo* (N.B.: vale anche per la funzione main!); il tipo di una funzione che non ha una dichiarazione esplicita di tipo è int

Il valore viene restituito da una istruzione `return(x);` il tipo di x deve essere consistente con quello di *func*.

Le funzioni debbono obbligatoriamente avere una lista di argomenti, eventualmente vuota

Non è possibile passare vettori, strutture o procedure come parametri, anche se è permesso il passaggio di puntatori a tutti questi tipi
(in realta' certi compilatori permettono di passare strutture)

I parametri delle funzioni vengono sempre passati per valore, ma siccome tali valori possono essere indirizzi di variabile: in Pascal equivale a passare una variabile per riferimento

I vettori sono passati per riferimento se si usa solamente il nome del vettore, senza indici;

se questa informazione è presente, come in `a[5]`, il riferimento al vettore rappresenta un singolo elemento e come tale è passato per valore;

si può usare l'operatore & per trasformare l'elemento in un puntatore all'elemento

Le funzioni (esempi)

Esempio

```
fil(buf, s, c)
   char *buf;
   int s;
   char c;
   {
   while (s--)
       *buf ++ = c;
   }
```

La funzione riempie un buffer, cui punta `buf`, con `s` copie del carattere `c`; termina senza restituire un valore, uscendo dal blocco principale

Il risultato della funzione è il contenuto del buffer cui punta buf

Input e output con specifica di formato

Esistono funzioni di libreria capaci di leggere e scrivere dati sotto il controllo di un formato specificabile dal programmatore: nelle invocazioni che seguono, la stringa *format* contiene alcuni specificatori frammisti a costanti stringa

Stampa
Gli argomenti *arg1*, *arg2* ecc. vengono stampati sullo standard output, oppure su un file arbitrario (stream), oppure su una stringa

```
printf(format, arg1, arg2, …);
fprintf(stream, format, arg1, arg2, …);
sprintf(string, format, arg1, arg2, …);
```

Lettura
I dati possono essere letti e messi in *arg1*, *arg2*, ecc. dallo standard input, da un canale arbitrario, oppure da una stringa

```
scanf(format, arg1, arg2, …);
fscanf(stream, format, arg1, arg2, …);
sscanf(string, format, arg1, arg2, …);
```

È obbligatorio che gli argomenti siano puntatori alle locazioni dove verrà memorizzato il dato; Inoltre i parametri di tipo stringa devono accordarsi con esattezza con i dati in arrivo sul canale

Esempi:

```
printf("%s", linea);
printf("%3d %6.1f\n",fahr,(5.0/9.0)*(fahr-32));
scanf("%1f",&v);
scanf("%d %s %d", @day, monthname, &year);
```

Struttura ed elaborazione di programmi C

Programma
Un programma consiste di uno o più file contenenti funzioni e dichiarazioni; le dichiarazioni di funzione **non** possono essere annidate

Variabili globali
È possibile dichiarare delle variabili globali, che non appartengono a nessuna funzione

Variabili esterne
Una funzione può accedere ad una variabile *v* (che non sia `static`), dichiarata in un file diverso da quello a cui appartiene

```
extern int v;
```

La dichiarazione `extern` serve semplicemente per informare il compilatore di che tipo sia la variabile: non viene allocata memoria

Ogni variabile globale deve essere dichiarata una sola volta senza l'attributo `extern`, perché le venga allocata la memoria necessaria

Precompilazione
Prima che un file sorgente venga compilato, viene automaticamente elaborato da un programma chiamato *precompilatore*

Il precompilatore esegue tre tipi di direttive
(le direttive iniziano con il carattere '#' in colonna 1):

- Inclusione di file
- Definizione ed espansione di macro
- Compilazione condizionale

Il precompilatore (continua)

Inclusione di file

È pratica comune raggruppare le dichiarazioni usate da diversi file in un *file di intestazione* (di solito col suffisso .h) e includere questo file dove servono quelle dichiarazioni

```
#include "file.h"
```

il file viene cercato nel directory di lavoro

```
#include <file.h>
```

il file viene cercato in */usr/include* invece che nel directory di lavoro

Macro

```
#define BLOCK_SIZE 1024
```

definisce una costante *BLOCK_SIZE* e le assegna il valore 1024

```
#define max(A,B) ((A) > (B) ? (A) : B))
```

definisce una macro max che calcola il maggiore tra due argomenti

```
#define forever for(;;)
```

definisce una macro `forever` che esegue ad un ciclo infinito

Compilazione condizionale

In MINIX ci sono diversi punti in cui il codice va compilato in modo diverso per CPU diverse

```
#ifdef i8088
    istruzioni per il processore 8088
#endif
```

Se il simbolo *i8088* è definito (ad esempio con `#define i8088`), allora le istruzioni tra le direttive vengono incluse nell'uscita del precompilatore; altrimenti, sono omesse

Il comando di compilazione

```
cc -c -Di8088 prog.c
```

forza la definizione del simbolo *i8088*

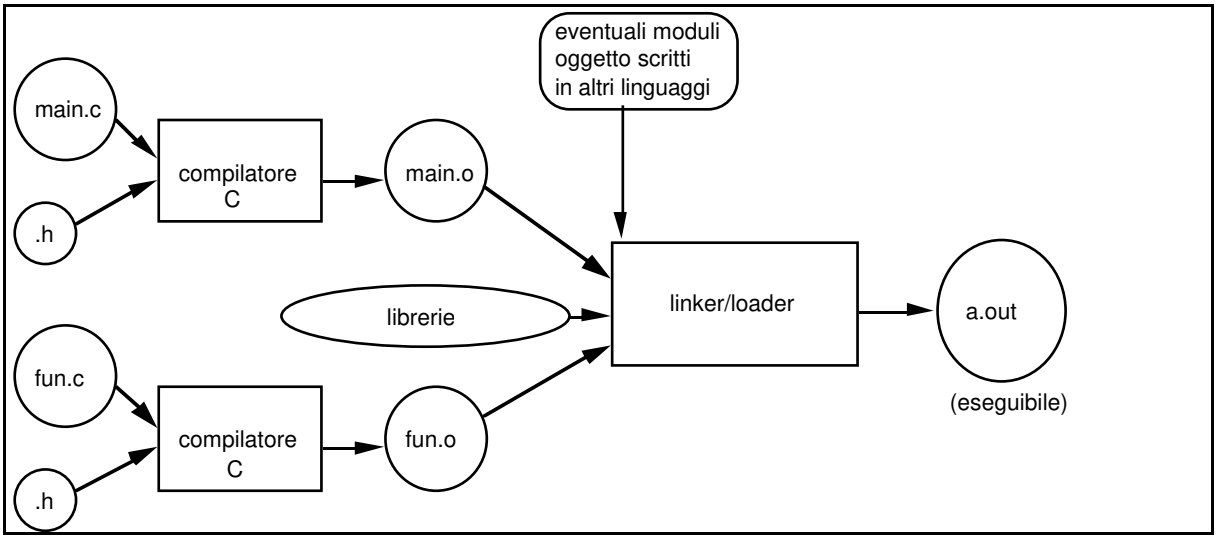
Il linker/loader

In C è ammessa la *compilazione separata*: ciascun file può essere compilato in momenti diversi, ottenendo un *modulo oggetto*
Otteniamo dunque tanti moduli oggetto separati, che in seguito legheremo insieme (mediante il linker)

Il compilatore vede un solo file alla volta: non ha la possibilità di catturare errori che presuppongono la conoscenza di più file; esiste uno strumento: LINT, che controlla i vincoli di linking

Per un linker un modulo oggetto è un insieme di riferimenti esterni:
ogni variabile e ogni funzione non dichiarate `static` sono esterne

Due riferimenti esterni con lo stesso nome sono proibiti entro lo stesso modulo oggetto; sono ammessi (ma solo per le variabili, non per le funzioni) se in moduli diversi: è il linker che gestisce i conflitti.



Programmazione C sotto Unix

Nel caso di programmi per Unix, la funzione `main` può avere due argomenti speciali chiamati `argc` e `argv`
Il primo è un intero che indica con quanti parametri è stato invocato il programma; Il secondo è un vettore di puntatori a stringhe che sono i parametri del programma

Esempio:
Il programma che segue conta il numero di caratteri in input; può operare in 2 modi, prolisso e sintetico, in funzione del valore della variabile **frugal**, che viene inizializzata dalla lista di parametri

```
# include <stdio>main(argc, argv)int argc;            /* Numero
di argomenti del programma */char *argv[];          /* Vettore
di puntatori */  {     long nc; char *s; int frugal = 0;   while
(--argc > 0 && (*++argv)[0] =='-')       {          /*
Elaborazione delle opzioni */            for (s = argv[0]+1 ; *s
!= '\0' ; s++)              switch (*s)                    {
          case 'f':                                 frugal =
TRUE;                  break;                          default:
             printf(''Opz. illegale\n'');
    break;             }}    nc = 0;     while (getchar() !=
EOF) ++nc; if (frugal) printf("%ld\n", nc);           else
    printf("Numero di caratteri = %ld\n", nc);     }
```

```
/* realizzazione del comando "cp src dest" : copia il file src
nel file dest */

#include <sys/types.h>   /* definizioni di tipo */
#include <fcntl.h>     /* definizione O_RDONLY, ecc. */
#include <stdlib.h>  /* contiene i prot. di chiamate di sistema */
#include <unistd.h>  /* contiene i prot. di chiamate di sistema */

void main(int argc, char *argv[]);  /* prototipo ANSI C */

#define BUF_SIZE 4096
#define MODE 0666

void main(int argc, char *argv[]);
/* argc: #argomenti; argv: puntatori agli argomenti */
{
   int src, dst, in, out;
   char buf[BUF_SIZE];

if (argc != 3) exit(1);   /* errato numero di argomenti */

   src= open(argv[1], O_RDONLY);    /* apri il file sorgente */
   if  (src < 0) exit(2);          /* errore: file non aperto */
   dst = creat(argv[2],MODE);   /* crea il file destinazione */
   if (dst < 0) exit(3);         /* errore: file non creato */

   while(1) {
      in=read(src,buf,BUF_SIZE);   /* leggi dal file sorgente */
      if (in <= 0) break;         /* esci ciclo se file finito */
      out = write(dst,buf,in);  /* scrivi sul file destinaz. */
      if (out <= 0) break;         /* esci ciclo se errore */
   }

   close(src);
   close(dst);
   exit(0);
}
```

L'ambiente di programmazione

L'editor
Emacs, Vi (sotto Unix)

La shell
Cshell
Bourne shell

Il compilatore
(`cc, lint`)

Il debugger simbolico
(Unix: `ctrace, dbx, adb`; MSDOS: `MS CodeView`)

La gestione della configurazione
(`make`)

La gestione degli accessi e delle versioni
(`SCCS`)

makefile (esempio)

```
all: client.o client.wic pshell.wic polis
     placeData

polis:  server.h server.cl child.cl
  clc -o polis server.cl child.cl

client.o: server.h client.c
  cc -c -o client.o client.c

client.wic: client.pro client.o
  BIMpcomp -c client.pro

place.wic: place.pro client.pro
  BIMpcomp -c place.pro

placeData: place.wic
  BIMlinker -h POLIS -r -HplaceData -o -
HplaceRun - -TCb10k place -TCb10k client -
LUnixFileSys

pshell.wic: server.h pshell.pro client.pro
client.o
  BIMpcomp -c pshell.pro

backup:
  cp README *.h *.pro *.cl *.c place
makefile sources

restore:
  cp sources/* .

cleanup:
  rm *.wic *.o polis
```