Programmare in C su Amiga (43)

di Dario de Judicibus

Terza puntata dedicata alle strutture dati dell'Amiga. In questa puntata continueremo la nostra analisi delle mappe di tastiera, e prenderemo da qui lo spunto per aprire una piccola parentesi sulla struttura interna dei file binari dell'Amiga. Parleremo di hunk, unità di programma, puntatori rilocabili ed altro. Entreremo in un mondo di bit e byte, nel cuore vero e proprio dei programmi, per capire un po' meglio che cosa è effettivamente un eseguibile e come fa a girare nel nostro Amiga

I tasti morti doppi

Abbiamo terminato la scorsa puntata con la promessa di parlare dei blocchi extra relativi ai *tasti morti doppi*. Vediamo di che si tratta. Per la definizione di tasto morto e di blocco extra fate riferimento all'articolo pubblicato lo scorso mese.

Un tasto morto doppio non è altro che un'estensione del concetto di tasto morto muto. La differenza è che un tasto morto muto modifica direttamente un tasto morto parlante premuto subito dopo, mentre un tasto morto doppio richiede la pressione prima di un altro tasto morto doppio, e solo in seguito quella di un tasto morto parlante. In pratica il carattere è generato dalla pressione *consecutiva* di ben tre tasti.

Un esempio è il tasto che nella tastiera tedesca contiene l'accento acuto e quello grave.

Se si preme il tasto da solo, non succede niente. L'accento acuto è quindi un tasto morto. Se a questo punto ripremete lo stesso tasto insieme a *shift*... non succede ancora niente. Si tratta allora di un tasto morto doppio. Infatti, se a questo punto premete il tasto con la **«A»**, il risultato finale sarà una **«â»**, cioè la vocale *a* con l'accento circonflesso. Quest'ultimo infatti si potrebbe pensare come formato da due sbarrette inclinate nei due sensi opposti: i due accenti appunto.

Un tasto morto doppio è definito nel modo seguente. Ricorderete sicuramente che il blocco extra di un tasto morto contiene il tipo di tasto (**0, DPF_DEAD** e **DPF_MOD**). Nel caso di un tasto morto muto (cioè **DPF_DEAD**), il secondo byte indica quale carattere va emesso tra quelli presenti nella tabella associata ad un qualsiasi tasto morto parlante. Esso infatti contiene l'indice del carattere da emettere all'interno della tabella in questione. Per indicare che un tasto morto è doppio è necessario aggiungere a questo indice un determinato valore spostato di un numero di bit uguale a **DP_2DFACSHIFT.** Questo valore corrisponde al numero totale di tasti morti muti più uno (inclusi quelli doppi). Per far questo basta utilizzare l'operatore di scorrimento a sinistra come segue:

DC.B DPF_DEAD, indice+((tasti morti muti +1)<<DP_2DFACSHIFT)

In figura 1 è riportata la logica utilizzata dal sistema per gestire i tasti morti doppi. In pratica, mentre per i tasti morti muti semplici la tabella associata ad ogni tasto morto parlante ha tante entrate quanti sono i tasti morti muti semplici, più una per il tasto non modificato, nel caso che esistano anche tasti morti doppi la cosa è un pochino più complicata.

Qui dobbiamo tener conto anche di tutte le possibili combinazioni in cui il primo tasto è un tasto morto qualunque, mentre il secondo è un tasto morto doppio.

Se ad esempio abbiamo tre tasti morti semplici e due doppi, avremo ben sei per tre caratteri in tabella, cioè diciotto in tutto. Un esempio è riportato in figura 2.

Da notare le seguenti cose:

- tutte le tabelle associate ad i tasti morti parlanti hanno la stessa lunghezza, che dipende dal numero di tasti morti muti in totale più uno (TotM+1) moltiplicato per il numero di tasti morti doppi più uno (TotD+1);
- ognuna di queste tabelle ha (TotD+1) righe per (TotM+1) colonne;
- l'indice parte sempre da zero;
- per ogni riga gli indici più bassi si riferiscono ai tasti morti doppi;
- la prima entrata di ogni riga corrisponde alle prime entrate della prima riga.

Per quanto riguarda l'ultimo punto lascio a voi scoprire il perché. Se ci riuscite vuol dire che non c'è blocco extra che possa fermarvi. Vi do un indizio. Se premete un solo tasto morto doppio prima di uno parlante, l'indice viene calcolato come se fossero stati premuti due tasti morti, ma non viene sommato (ovviamente) l'indice del primo dei due (visto che ce n'è uno solo).

La tastiera italiana

A questo punto abbiamo tutti gli elementi per analizzare in dettaglio la ta-

stiera italiana. In figura 3 è riportato il file **devs:keymaps/i** in formato esadecimale.

Che ve ne sembra? Ci capite qualcosa? Beh, non so voi, ma onestamente la prima volta che stampai questo file cinque anni fa, ci misi svariate ore per distinguervi alcune delle strutture che vi ho descritto in queste ultime puntate, ed anche così rimanevano fuori un bel po' di byte. E allora? In effetti ci manca ancora qualcosa. Prima di vedere cosa, provate a fare il seguente esercizio.

Mettetevi sul vostro Amiga, scrivete il seguente file

```
#define NONE
```

e chiamatelo **test.c**. Compilatelo senza alcuna opzione. Ad esempio, nel caso del *SAS/C 5.1* lanciate il comando **lc test.c**. Il risultato sarà un file chiamato **test.o**. Lanciare il *linker* ora, *senza* specificare alcuna libreria di compilazione e senza fornire il nome di un file di partenza [*startup file*] come ad esemio **c.o**. Sempre nel caso del *SAS/C* scriveremo dunque **blink test.o**. Il risultato sarà il file **test**.

Andate a vedere la struttura interna di questo fle con il comando **type test**

◀ *Figura 1*
Logica di gestione dei tasti morti doppi.

```
INIZIO

  Viene premuto un tasto morto parlante

  SE
    Il tasto precedente è un tasto morto muto
  ALLORA
    Vai alla sezione A:
  ALTRIMENTI
    Emetti il carattere corrispondente al tasto non modificato

FINE

A: INIZIO

  SE
    Anche il secondo tasto precedente è un tasto morto muto
  ALLORA
    Vai alla sezione B:
  ALTRIMENTI
    Vai alla sezione C:

RITORNA

B: INIZIO

  SE
    E' un tasto morto doppio
  ALLORA
    Estrai l'indice del secondo tasto morto muto e moltiplicalo
    per il numero di tasti morti più uno
    Aggiungi al valore ottenuto l'indice del primo tasto morto
    muto
    Usa il risultato come indice finale nella tabella associata
    al tasto morto parlante

RITORNA

C: INIZIO

  SE
    E' un tasto morto doppio
  ALLORA
    Estrai l'indice del tasto morto muto e moltiplicalo
    per il numero di tasti morti più uno
    Usa il risultato come indice finale nella tabella associata
    al tasto morto parlante
  ALTRIMENTI
    Utilizza l'indice del tasto morto muto per identificare il
    carattere da emettere nella tabella associata al tasto morto
    parlante

RITORNA
```

Figura 2
Tabella di emissione per i tasti morti.
▼

```
  mp  tasto morto parlante

TASTI MUTI DOPPI

  d1    accento acuto          ´
  d2    accento grave          `

TASTI MUTI SEMPLICI

  s1    accento circunflsso    ^
  s2    tilde                  ~
  s3    virgolette             "

CARATTERI

  0x61  a minuscola            a
  0xE0  a aacuta               á
  0xE1  a grave                à
  0xE2  a circonflessa         â
  0xE3  a tilde                ã
  0xE4  a umlaut               ä

TABELLA RELATIVA AL TASTO "a"
```

1° \ 2°	mp	d1	d2	s1	s2	s3
mp	61	E1	E0	E2	E3	E4
d1	E1	E1	E2	E1	E1	E1
d2	E0	E2	E0	E0	E0	E0

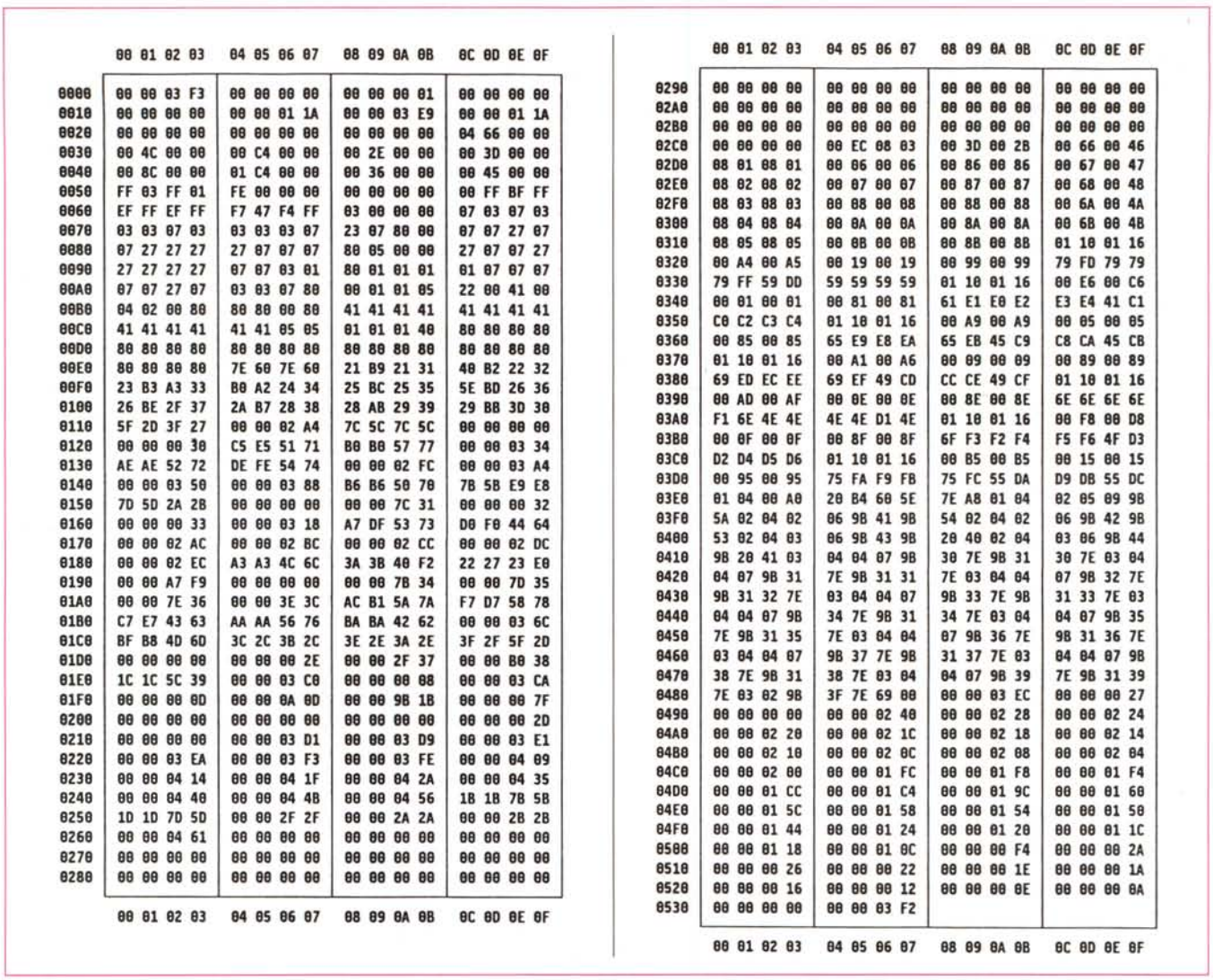
```
ESEMPIO:

se si preme in successione

d1 d2 mp

avremo:

indice di d2 (2) per numero
di tasti morti più uno (6)
più indice di d1 (1) -> 13
cioè 0xE2 (si parte da 0).
```



	00 01 02 03	04 05 06 07	08 09 0A 0B	0C 0D 0E 0F
0000	00 00 03 F3	00 00 00 00	00 00 00 01	00 00 00 00
0010	00 00 00 00	00 00 01 1A	00 00 03 E9	00 00 01 1A
0020	00 00 00 00	00 00 00 00	00 00 00 00	04 66 00 00
0030	00 4C 00 00	00 C4 00 00	00 2E 00 00	00 3D 00 00
0040	00 8C 00 00	01 C4 00 00	00 36 00 00	00 45 00 00
0050	FF 03 FF 01	FE 00 00 00	00 00 00 00	00 FF BF FF
0060	EF FF EF FF	F7 47 F4 FF	03 00 00 00	07 03 07 03
0070	03 03 07 03	03 03 03 07	23 07 80 00	07 07 27 07
0080	07 27 27 27	27 07 07 07	80 05 00 00	27 07 07 27
0090	27 27 27 27	07 07 03 01	80 01 01 01	01 07 07 07
00A0	07 07 27 07	03 03 07 80	00 01 01 05	22 00 41 00
00B0	04 02 00 80	80 80 00 80	41 41 41 41	41 41 41 41
00C0	41 41 41 41	41 41 05 05	01 01 01 40	80 80 80 80
00D0	80 80 80 80	80 80 80 80	80 80 80 80	80 80 80 80
00E0	80 80 80 80	7E 60 7E 60	21 B9 21 31	40 B2 22 32
00F0	23 B3 A3 33	B0 A2 24 34	25 BC 25 35	5E BD 26 36
0100	26 BE 2F 37	2A B7 28 38	28 AB 29 39	29 BB 3D 30
0110	5F 2D 3F 27	00 00 02 A4	7C 5C 7C 5C	00 00 00 00
0120	00 00 00 30	C5 E5 51 71	B0 B0 57 77	00 00 03 34
0130	AE AE 52 72	DE FE 54 74	00 00 02 FC	00 00 03 A4
0140	00 00 03 50	00 00 03 88	B6 B6 50 70	7B 5B E9 E8
0150	7D 5D 2A 2B	00 00 00 00	00 00 7C 31	00 00 00 32
0160	00 00 00 33	00 00 03 18	A7 DF 53 73	D0 F0 44 64
0170	00 00 02 AC	00 00 02 BC	00 00 02 CC	00 00 02 DC
0180	00 00 02 EC	A3 A3 4C 6C	3A 3B 40 F2	22 27 23 E0
0190	00 00 A7 F9	00 00 00 00	00 00 7B 34	00 00 7D 35
01A0	00 00 7E 36	00 00 3E 3C	AC B1 5A 7A	F7 D7 58 78
01B0	C7 E7 43 63	AA AA 56 76	BA BA 42 62	00 00 03 6C
01C0	BF B8 4D 6D	3C 2C 3B 2C	3E 2E 3A 2E	3F 2F 5F 2D
01D0	00 00 00 00	00 00 00 2E	00 00 2F 37	00 00 B0 38
01E0	1C 1C 5C 39	00 00 03 C0	00 00 00 00	00 00 03 CA
01F0	00 00 00 0D	00 00 0A 0D	00 00 9B 1B	00 00 00 7F
0200	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 2D
0210	00 00 00 00	00 00 03 D1	00 00 03 D9	00 00 03 E1
0220	00 00 03 EA	00 00 03 F3	00 00 03 FE	00 00 04 09
0230	00 00 04 14	00 00 04 1F	00 00 04 2A	00 00 04 35
0240	00 00 04 40	00 00 04 4B	00 00 04 56	1B 1B 7B 5B
0250	1D 1D 7D 5D	00 00 2F 2F	00 00 2A 2A	00 00 2B 2B
0260	00 00 04 61	00 00 00 00	00 00 00 00	00 00 00 00
0270	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0280	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

	00 01 02 03	04 05 06 07	08 09 0A 0B	0C 0D 0E 0F
0290	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
02A0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
02B0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
02C0	00 00 00 00	00 EC 08 03	00 3D 00 2B	00 66 00 46
02D0	08 01 08 01	00 06 00 06	00 86 00 86	00 67 00 47
02E0	08 02 08 02	00 07 00 07	00 87 00 87	00 68 00 48
02F0	08 03 08 03	00 08 00 08	00 88 00 88	00 6A 00 4A
0300	08 04 08 04	00 0A 00 0A	00 8A 00 8A	00 6B 00 4B
0310	08 05 08 05	00 0B 00 0B	00 8B 00 8B	01 10 01 16
0320	00 A4 00 A5	00 19 00 19	00 99 00 99	79 FD 79 79
0330	79 FF 59 DD	59 59 59 59	01 10 01 16	00 E6 00 C6
0340	00 01 00 01	00 81 00 81	61 E1 E0 E2	E3 E4 41 C1
0350	C0 C2 C3 C4	01 10 01 16	00 A9 00 A9	00 05 00 05
0360	00 85 00 85	65 E9 E8 EA	65 EB 45 C9	C8 CA 45 CB
0370	01 10 01 16	00 A1 00 A6	00 09 00 09	00 89 00 89
0380	69 ED EC EE	69 EF 49 CD	CC CE 49 CF	01 10 01 16
0390	00 AD 00 AF	00 0E 00 0E	00 8E 00 8E	6E 6E 6E 6E
03A0	F1 6E 4E 4E	4E 4E D1 4E	01 10 01 16	00 F8 00 D8
03B0	00 0F 00 0F	00 8F 00 8F	6F F3 F2 F4	F5 F6 4F D3
03C0	D2 D4 D5 D6	01 10 01 16	00 B5 00 B5	00 15 00 15
03D0	00 95 00 95	75 FA F9 FB	75 FC 55 DA	D9 DB 55 DC
03E0	01 04 00 A0	20 B4 60 5E	7E A8 01 04	02 05 09 9B
03F0	5A 02 04 02	06 9B 41 9B	54 02 04 02	06 9B 42 9B
0400	53 02 04 03	06 9B 43 9B	20 40 02 04	03 06 9B 44
0410	9B 20 41 03	04 04 07 9B	30 7E 9B 31	30 7E 03 04
0420	04 07 9B 31	7E 9B 31 31	7E 03 04 04	07 9B 32 7E
0430	9B 31 32 7E	03 04 04 07	9B 33 7E 9B	31 33 7E 03
0440	04 04 07 9B	34 7E 9B 31	34 7E 03 04	04 07 9B 35
0450	7E 9B 31 35	7E 03 04 04	07 9B 36 7E	9B 31 36 7E
0460	03 04 04 07	9B 37 7E 9B	31 37 7E 03	04 04 07 9B
0470	38 7E 9B 31	38 7E 03 04	04 07 9B 39	7E 9B 31 39
0480	7E 03 02 9B	3F 7E 69 00	00 00 03 EC	00 00 00 27
0490	00 00 00 00	00 00 02 40	00 00 02 28	00 00 02 24
04A0	00 00 02 20	00 00 02 1C	00 00 02 18	00 00 02 14
04B0	00 00 02 10	00 00 02 0C	00 00 02 08	00 00 02 04
04C0	00 00 02 00	00 00 01 FC	00 00 01 F8	00 00 01 F4
04D0	00 00 01 CC	00 00 01 C4	00 00 01 9C	00 00 01 60
04E0	00 00 01 5C	00 00 01 58	00 00 01 54	00 00 01 50
04F0	00 00 01 44	00 00 01 24	00 00 01 20	00 00 01 1C
0500	00 00 01 18	00 00 01 0C	00 00 00 F4	00 00 00 2A
0510	00 00 00 26	00 00 00 22	00 00 00 1E	00 00 00 1A
0520	00 00 00 16	00 00 00 12	00 00 00 0E	00 00 00 0A
0530	00 00 00 00	00 00 03 F2		

Figura 3 - Tastiera Keymaps/i.

hex. Il risultato sarà quello riportato in figura 4. Notate niente?

Provate a confrontare la mappa di tastiera con questo file.

Concentratevi soprattutto sulle prime due righe della mappa di tastiera e sull'ultima.

Siete ancora perplessi? OK, andiamo avanti...

La struttura dei file binari

Prima di proseguire con l'analisi della nostra mappa di tastiera, è necessario a questo punto aprire una piccola parentesi.

Vi siete mai chiesti come è fatto un programma, *dentro?* Voglio dire, tutti voi sapete come scrivere un programma, come si compila e si «ricuce» [*link*]. Ma sapete il perché di questi vari passi e qual è la struttura dell'eseguibile alla fine di tutto il processo? Come fa l'Amiga ad eseguire un programma?

Fintanto che non sapremo rispondere a queste domande non potremo iniziare l'analisi binaria della mappa di tastiera. Non vedremo la struttura completa dei file binari dell'Amiga, ma solo i blocchi principali e quelli che ci serviranno per capire come è fatto internamente il file **keymaps/i**.

Alcuni concetti base

Vediamo innanzitutto di definire alcuni termini che useremo in seguito.

Quando compilate un sorgente il risultato è quello che si chiama un *file oggetto* [*object file*]. Questo file in genere contiene tutta una serie di riferimenti ad oggetti esterni, detti appunto *riferimenti esterni* [*external references*]. Questi riferimenti sono memorizzati nel file og-

	00 01 02 03	04 05 06 07	08 09 0A 0B	0C 0D 0E 0F
0000	00 00 03 F3	00 00 00 00	00 00 00 01	00 00 00 00
0010	00 00 00 00	00 00 00 00	00 00 03 E9	00 00 00 00
0020	00 00 03 F2			

Figura 4
File di prova TEST.

getto utilizzando il *nome* dell'oggetto referenziato. Questo oggetto può essere una funzione in una libreria esterna, una variabile, un comando residente, e così via. È evidente che fintanto che questi riferimenti non vengono risolti, il file oggetto non potrà mai essere eseguito dal sistema operativo. Supponiamo ad esempio che il mio programma chiami una funzione **pippo()** non contenuta nel file stesso. Come fa l'Amiga a sapere dove si trova **pippo()**? Magari c'è una funzione con quel nome in una libreria residente, od in un altro file oggetto, od in una libreria di compilazione [*compile-time library*] in **LIBS:**. Ma qual è quella giusta?

Per risolvere tutte le referenze esterne è necessario utilizzare un altro programma, detto *linker* od anche *linkage editor*, che in inglese significa «ciò che unisce, collega, concatena». Questo programma riceve in ingresso tutti i file oggetto, le librerie di compilazione, il file di caricamento [*startupfile*] e tutte le opzioni ed informazioni necessarie per risolvere tutte le referenze esterne. Questo vuol dire che ad ogni nome contenuto nei vari file oggetto, viene sostituito un puntatore al pezzo di codice, all'area di memoria, od al blocco di dati corrispondente.

A questo punto abbiamo il nostro *eseguibile* [*load file*]. Esso è formato di uno o più blocchi di byte chiamati *hunk*, che significa «grosso pezzo». Vi sono diversi tipi di *hunk*, ognuno con una struttura ben definita. Più avanti ne vedremo alcuni. Gli *hunk* sono raggruppati in *unità di programma* [*program unit*], che rappresentano l'elemento atomico che un *linker* è in grado di gestire. La relazione che lega le unità di programma alla struttura del codice sorgente può variare da compilatore a compilatore.

Un discorso importante riguarda i puntatori. Questi possono essere *assoluti* o *rilocabili*. Un puntatore si dice assoluto quando contiene il valore effettivo dell'indirizzo di memoria nel sistema a cui si riferisce. Ad esempio, un registro hardware sarà referenziato in modo assoluto. Lo stesso dicasi per una funzione residente nella ROM.

Viceversa, un puntatore che punta ad un altro elemento dello stesso eseguibile non potrà contenere il vero indirizzo di quell'elemento una volta che il programma è stato caricato in memoria. Questi infatti dipenderà dall'indirizzo a partire dal quale l'eseguibile è stato caricato, indirizzo che addirittura in certi sistemi operativi può variare *durante l'esecuzione* del programma stesso. In questo caso il puntatore conterrà la distanza che c'è tra l'elemento suddetto e l'inizio del programma, od un qualunque altro punto predefinito, a seconda del sistema operativo. Inoltre il *linker* si assumerà il compito di mantenere una tabella che contiene la posizione di tutti i puntatori rilocabili, in modo che il sistema operativo possa modificarli opportunamente quando il programma viene mandato in esecuzione. Una volta, quando non esisteva il concetto di rilocabilità, tutti i programmi dovevano partire dalla locazione **zero** della memoria di sistema. Ed in effetti, è proprio quello che succede ancora in fase di partenza [*bootstrap*] del cuore [*kernel*] della maggior parte dei sistemi operativi.

Gli hunks

Vediamo adesso alcuni *hunk* che ci serviranno in seguito.

Abbiamo detto che ce ne sono vari tipi. In linea di massima possiamo defi-

◀ *Figura 5*
Hunks.

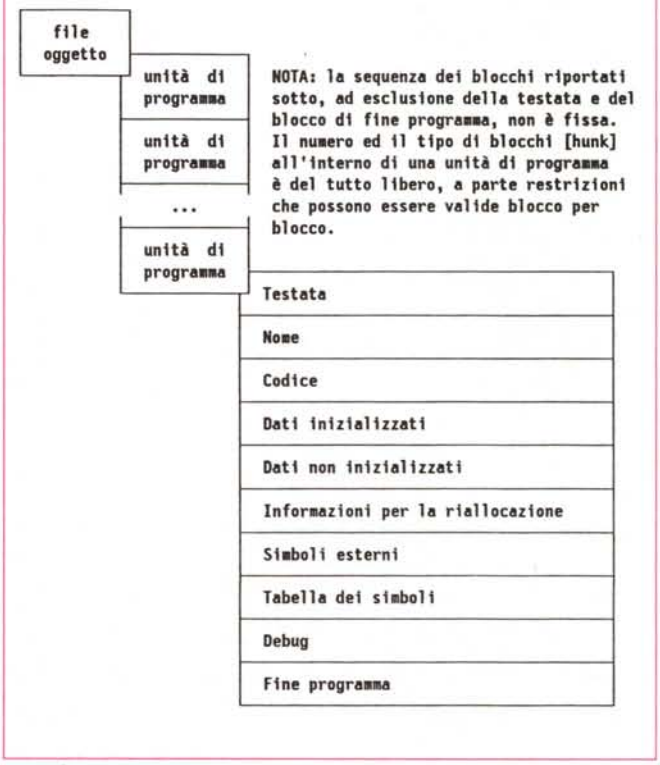
FILE OGGETTO

Tipo hunk	Codice decimale	Codice esadecimale
HUNK_UNIT	999	3E7
HUNK_NAME	1000	3E8
HUNK_CODE	1001	3E9
HUNK_DATA	1002	3EA
HUNK_BSS	1003	3EB
HUNK_RELOC32	1004	3EC
HUNK_RELOC16	1005	3ED
HUNK_RELOC8	1006	3EE
HUNK_EXT	1007	3EF
HUNK_SYMBOL	1008	3F0
HUNK_DEBUG	1009	3F1
HUNK_END	1010	3F2

ESEGUIBILE

Tipo hunk	Codice decimale	Codice esadecimale
HUNK_HEADER	1011	3F3
HUNK_OVERLAY	1013	3F5
HUNK_BREAK	1014	3F6

Figura 6
La struttura di un file oggetto.
▼



nire quattro classi di *hunk:*
1. quelli che contengono informazioni per il sistema,
2. quelli contenenti il codice esguibile, o *code hunks*,
3. quelli contenenti i dati inizializzati, o *data hunks*, e
4. quelli contenenti i dati *non* inizializzati, o *bss hunks*.

Consideriamo prima i file oggetto.

Un file oggetto è formato da una o più unità di programma. Ogni unità è formata da una testata, e da uno o più *hunk*. In figura 6 è riportata la struttura di un file oggetto, ed i vari tipi di *hunk* che possono essere contenuti in ciascuna unità di programma. La figura riportata non indica tuttavia la sequenza ed il numero preciso di *hunk* che sono contenuti in una unità di programma, ma solo il tipo di *hunk* che essa può contenere.

Le uniche eccezioni riguardano la testata ed il blocco di fine programma, ovviamente.

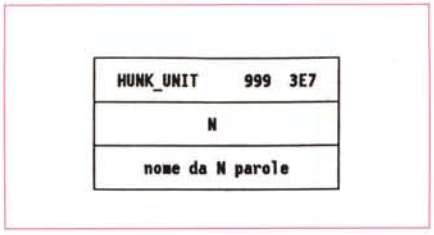


Figura 7 - Hunk_unit.

Per quello che riguarda gli eseguibili, essi hanno una struttura molto simile a quella dei file oggetto, con le sole seguenti eccezioni:
- non contengono alcuna informazione relativa alle unità di programma,
- non possono contenere un blocco per i simboli esterni,
- tutti i simboli esterni sono stati risolti,
- tutti i blocchi di riallocazione sono del tipo a 32 bit,
- iniziano sempre con un *hunk* che riporta il numero di *hunk* che va caricato in memoria.

Ci sarebbero altri aspetti della struttura degli eseguibili, relativi alle tecniche di *overlay*, che vanno per il momento al di là degli scopi di questa breve parentesi sulle strutture dei file binari dell'Amiga.

In figura 5 sono riportati *tutti* gli *hunk* relativi sia ai file oggetto, che agli eseguibili, completi di codice di identificazione, sia in valore decimale che esadecimale. Noi ora ne vedremo in dettaglio solo alcuni che ci serviranno in seguito.

hunk_unit

È il blocco che identifica l'inizio di una nuova unità di programma, detto anche *testata* [*header block*].

Esso è formato da una parola (quattro byte) contenente il codice di identifica-

La scheda tecnica: Inside 2.0

In questa puntata vedremo altre cinque schede tecniche relative a funzioni della **gadtools.library**.

CreateMenusA

Alloca e riempie una struttura per la definizione dei menu.

prototipo

```
struct Menu *CreateMenusA  // Puntatore alla struttura Menu completa
(
  struct NewMenu *newmenu, // Puntatore ad una serie di struct NewMenu
  struct TagItem *taglist  // Puntatore ad una lista di "tag"
) ;
```

A seconda del contenuto delle varie strutture **NewMenu** fornite in ingresso, questa funzione è in grado di creare e riempire correttamente tutte le strutture necessarie alla definizione di menu, voci e sottovoci e di gestire i legami tra le stesse.

Qualora non fosse possibile allocare la memoria necessaria, o nel caso la lista dei vari «tag» contenesse errori di sintassi, **CreateMenusA()** fornirà indietro un valore nullo al posto del puntatore alla struttura **Menu** che rappresenta la radice di tutte le strutture necessarie a definire i menu e le eventuali voci e sottovoci richieste. Qui di seguito è riportata la scheda relativa ai vari *tag* validi per questa funzione.

Tag / Tipo	Descrizione
GTMN_FrontPen UBYTE	Numero della penna da utilizzare per il testo del menù (il default è 0).
GTMN_FullMenu BOOL	Se TRUE, la serie di strutture NewMenu descrive un menù completo. Se FALSE, descrive solo un frammento (default).
GTMN_SecondaryError ULONG *	Puntatore ad un'area di tipo LONG inizializzata a zero, ed intesa a fornire un campo in cui ricevere un codice descrittivo di errore.

Note:

1. Le varie stringhe fornite come testi dei menu e delle voci da creare non sono copiate, e quindi bisogna evitare di cancellarle per tutta la vita del menu.
2. Se **GTMN_FullMenu** == **TRUE** ma la serie di strutture **NewMenu** descrive solo un menu parziale (o frammento), la funzione ritorna il codice di errore **GTMENU_INVALID.**
3. Questa funzione crea automaticamente per tutti i menu, voci o sottovoci un opportuno campo **UserData** che può essere acceduto tramite apposite macro, riportate qui a seguire:

```
#define GTMENU_USERDATA(menu)  (* ( (APTR *)(((struct Menu *)menu)+1) ) )

#define GTMENUITEM_USERDATA(menuitem)  (* ( (APTR *)(((struct MenuItem *)menuitem)+1) ) )
```

4. I codici descrittivi di errore, o codici di condizione [*condition codes*], forniscono ulteriori informazioni sulle cause di un eventuale errore verificatosi. Una lista per la funzione suddetta è riportata qui sotto:

```
GTMENU_INVALID   La struttura NewMenu non descrive correttamente un
                 menù. La funzione fallisce.

GTMENU_TRIMMED   La struttura NewMenu contiene troppi menù, voci e
                 sottovoci. Viene creata solo una parte del menù
                 richiesto.

GTMENU_NOMEM     Non c'è abbastanza memoria per creare il menù
                 richiesto.
```

CreateMenus

Alloca e riempie una struttura per la definizione dei menu. Versione a parametri variabili della **CreateMenusA()**.

prototipo

```
struct Menu *CreateMenus   // Puntatore alla struttura Menu completa
(
  struct NewMenu *newmenu  , // Puntatore ad una serie di struct NewMenu
  (tagtype)       firsttag , // Primo tag
  ...
) ;
```

zione, la lunghezza in parole del nome dell'unità di programma, ed il nome stesso, allineato in fondo ad un numero intero di parole utilizzando degli zeri. La struttura è riportata in figura 7.

hunk_code

È un blocco, solitamente rilocabile, che contiene codice. La struttura è riportata in figura 8.

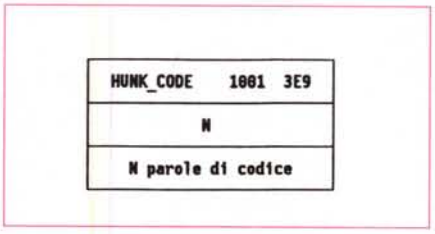


Figura 8 - Hunk_code.

hunk_reloc32

È un blocco utilizzato per definire quali informazioni vanno rilocate in un indirizzamento a 32 bit.

Dato che ogni *hunk* in una unità di programma è numerato a partire da zero, questo blocco contiene, a parte ovviamente il suo codice di identificazione, una sezione per ogni blocco rilocabile. Ogni sezione contiene il numero di indirizzi relativi ai puntatori da rilocare in un certo blocco, il numero sequenziale che identifica tale blocco, ed una serie di indirizzi relativi all'inizio del blocco [*offset*] che corrispondono a tutti i puntatori che in quel blocco vanno rilocati. Il blocco termina con una parola nulla. La struttura è riportata in figura 9.

hunk_end

Questo *hunk* segnala la fine dell'unità di programma, e contiene solo il suo codice di identificazione.

hunk_header

Questo *hunk* contiene una serie di informazioni relative al numero di *hunk* che va caricato in memoria, più il nome di ogni libreria residente che va caricata insieme a ciascun blocco di *hunk*. Tali nomi sono contenuti nella prima parte dell'*hunk*. Ogni nome è formato da una parola che ne dà la lunghezza in parole da quattro byte, ed il nome stesso con un certo numero di byte nulli in coda in modo da far sì che la lunghezza sia effettivamente un numero intero di parole. Questa lista termina con una parola nulla, ed è ordinata nella sequenza di caricamento delle librerie. Quando il sistema carica il programma, esso alloca una tabella che serve a tener traccia di tutti gli *hunk* caricati, inclusi quelli di *overlay*. La seconda parte dell'*hunk* contiene la dimensione massima che deve avere questa tabella, che è poi uguale al

Si tratta praticamente della **CreateMenusA()**, solo che invece di richiedere in ingresso come secondo parametro il puntatore ad una lista di *tag*, permette di introdurre direttamente i vari tag ed i valori loro eventualmente associati, secondo la tipica sintassi a parametri variabili (quella della **printf()**, tanto per intenderci).

DrawBevelBoxA

Disegna un rettangolo a sbalzo.

prototipo

```
VOID DrawBevelBoxA // Non ritorna alcun valore
(
  struct RastPort *rport    , // Dove va disegnato il rettangolo
  WORD             left     , // Ascissa dell'origine del rettangolo
  WORD             top      , // Ordinata dell'origine del rettangolo
  WORD             width    , // Larghezza del rettangolo
  WORD             height   , // Altezza del rettangolo
  struct TagItem   *taglist   // Lista di "tag"
) ;
```

Nota: un rettangolo a sbalzo non è un controllo, ma solo un elemento grafico, che va opportunamente restaurato [*refresh*] qualora venga danneggiato da una qualche operazione, come lo spostamento od il ridimensionamento del *raster* interessato.

Qui di seguito è riportata la scheda relativa ai vari *tag* validi per questa funzione.

Tag / Tipo	Descrizione
GTBB_Recessed BOOL	Se TRUE, allora il rettangolo sembrerà rientrare nel piano, se FALSE ne verrà fuori (default)
GT_VisualInfo APTR	Qui va messo, OBBLIGATORIAMENTE, il valore ottenuto da una precedente chiamata alla GetVisualInfoA().

DrawBevelBox

Disegna un rettangolo a sbalzo. Versione a parametri variabili della **DrawBevelBoxA()**.

prototipo

```
VOID DrawBevelBox // Non ritorna alcun valore
(
  struct RastPort *rport    , // Dove va disegnato il rettangolo
  WORD             left     , // Ascissa dell'origine del rettangolo
  WORD             top      , // Ordinata dell'origine del rettangolo
  WORD             width    , // Larghezza del rettangolo
  WORD             height   , // Altezza del rettangolo
  (tagtype)        firsttag , // Primo tag
  ...
) ;
```

Si tratta praticamente della **DrawBevelBoxA()**, solo che invece di richiedere in ingresso come sesto parametro il puntatore ad una lista di *tag*, permette di introdurre direttamente i vari tag e i valori loro eventualmente associati, secondo la tipica sintassi a parametri variabili (quella della **printf()**, tanto per intenderci).

FreeGadgets

Dealloca una lista concatenata di controlli.

prototipo

```
VOID FreeGadgets  // Non ritorna alcun valore
(
  struct Gadget *glist  // Puntatore al primo controllo nella lista
) ;
```

Dealloca tutti i controlli di tipo *GadTools* a partire da quello di cui viene fornito in ingresso il puntatore.

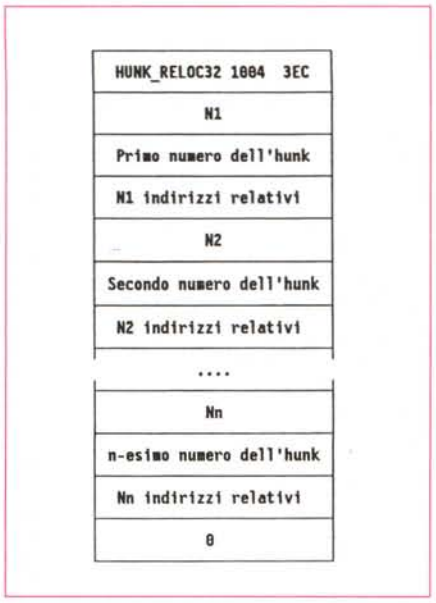


Figura 9 - Hunk_reloc32.

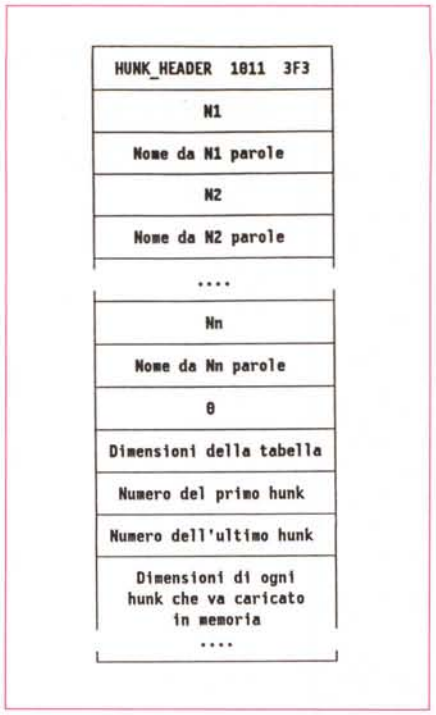


Figura 10 - Hunk_header.

numero massimo di *hunk* caricabili. A seguire c'è il numero d'ordine del primo *hunk* da caricare. Questi è zero se non ci sono librerie residenti da caricare, altrimenti corrisponde al numero totale di *hunk* di tutte le librerie in questione. Quindi c'è il numero d'ordine dell'ultimo *hunk* da caricare, per cui, se escludiamo le librerie residenti, l'eseguibile viene a contenere tanti *hunk* quanti sono dati sottraendo il numero d'ordine del primo blocco da quello dell'ultimo ed aggiungendo uno. Questo è anche il numero del vettore posto alla fine dell'**hunk_header**, che contiene le dimensioni di ogni singolo *hunk*. La struttura è riportata in figura 10.

Alla ricerca della mappa... di tastiera

E torniamo ora alla mappa di tastiera italiana, ed al programmino di prova che abbiamo scritto all'inizio, cioè **test.**

La prima parola di entrambi è **000003F3**, cioè il codice di identificazione dell'*hunk_header*. Ci aspettiamo allora che le parole successive contengano la lista delle librerie residenti da caricare, chiusa in fondo da una parola nulla. Ed in effetti la parola successiva è in entrambi i casi uno zero. E questo torna bene con il fatto che non abbiamo specificato alcuna libreria nel costruire **test**, e che non ci aspettiamo niente di diverso per un file che non dovrebbe rappresentare altro che una mappa di tastiera. A questo punto dovremmo avere il numero massimo di *hunk* caricabili. Ed in effetti la terza parola contiene **00000001**, cioè *uno*. Anche questo torna in entrambi i casi. Ancora, la quarta e la quinta parola dovrebbero riportare il numero d'ordine rispettivamente del primo e dell'ultimo *hunk* da caricare.

Dato che abbiamo un solo *hunk* nel file, nessuna libreria residente e nessun *hunk* di *overlay*, questi è *zero* in entrambi i casi. Per finire, la sesta parola dovrebbe dare la dimensione dell'unico *hunk* da caricare. E qui abbiamo **0000011A** per la tastiera italiana e zero per il file di prova, come è da aspettarsi visto che in effetti abbiamo creato un «eseguibile» vuoto. Per quello che riguarda la mappa di tastiera, ci aspettiamo quindi che contenga un blocco da 282 parole da quattro byte l'una, cioè 70 righe da quattro parole più due parole. Vedremo tra poco qual è. In figura 11 è riportata l'analisi del primo *hunk* di entrambi i file.

La parola che segue contiene **000003E9**, che corrisponde al codice di identificazione di un *hunk_code*. I quattro byte successivi conterranno quindi la dimensione di questo *hunk*. Questa è differente nei due casi, ovviamente. Per quello che riguarda il programma di prova, bisogna tener presente che esso non contiene altro che una istruzione per il preprocessore del C, la quale non genera affatto codice. L'ottava parola è quindi nulla per **test.** La mappa di tastiera invece, è contenuta tutta in questo singolo *hunk*, per cui ritroveremo qui le dimensioni dell'unico blocco del file **keymaps/i**, e cioè **0000011A** (282 parole).

Saltiamo ora per un momento all'ultima parola di entrambi i file. Questa è ovviamente *hunk_end*, e cioè **000003F2.**

L'analisi di **test** è quindi completa. Non così è invece per la tastiera italiana. Abbiamo visto che le prime sei parole fanno parte di un *hunk_header*, mentre la settima e l'ottava sono la prima parte di un *hunk_code* lungo 282 parole. Il corpo vero e proprio di questo *hunk* va quindi contato a partire dalla nona parola inclusa. Se contiamo 282 parole, arriviamo alla posizione **0x0488** non inclusa, come riportato in figura 3. La parola che parte da questa posizione contiene il codice **000003EC**, che guarda caso è il codice di identificazione di un *hunk_reloc32*. Ed in effetti la parola successiva riporta il valore esadecimale **00000027**, che corrisponde al numero di puntatori da rilocare nell'*hunk* numero zero, come indicato dalla parola seguente. Se ora contiamo 39 parole a partire dalla posizione **0x0494** inclusa, arriviamo ad un'altra parola nulla che è appunto l'indicatore di fine blocco dell'*hunk_reloc32*.

Per cui, per concludere, la mappa di tastiera vera e propria è compresa tra la posizione **0x0020** e **0x487** incluse.

Figura 11 Hunk_header per I e TEST.

I	DESCRIZIONE	TEST
00 00 03 F3	hunk_header	00 00 03 F3
00 00 00 00	lista delle librerie vuota	00 00 00 00
00 00 00 01	numero di hunk caricabili	00 00 00 01
00 00 00 00	primo hunk da caricare	00 00 00 00
00 00 00 00	ultimo hunk da caricare	00 00 00 00
00 00 01 1A	dimensioni dell'hunk	00 00 00 00

Conclusione

Nella prossima puntata concluderemo l'analisi in dettaglio di quest'area, nel tentativo di riconoscere le varie strutture descritte nelle precedenti puntate. Arrivederci tra un mese. MC

Dario de Judicibus è raggiungibile tramite Mc-link alla casella MC2120.