Programmare in C su Amiga

di Dario de Judicibus

Il nostro slalom parallelo continua con le tecniche avanzate di LMK *(fornito dalla Lattice nella versione 5 del suo compilatore C) e con la sezione grafica, e precisamente con la seconda parte dedicata alle funzioni di riempimento*

Nella 14ª puntata, abbiamo approfondito l'analisi del programma di utilità *LMK* e delle macro interne. Abbiamo inoltre iniziato a parlare di riempimenti di aree grafiche ed abbiamo visto un esempio di riempimento a *macchia d'olio*. In questa puntata continueremo entrambi i discorsi e vedremo in particolare, per quello che riguarda *LMK*:

- definizioni per bersagli multipli,
- definizioni per bersagli alternativi,
- le regole di default, ed
- i caratteri speciali.

Mentre, per quello che riguarda la libreria grafica, parleremo del:

- riempimento a definizione di area.

LMK

Continuiamo quindi il nostro discorso su *LMK* affrontando la parte relativa alle definizioni *avanzate*. Quanto detto fin qui su *LMK* sarebbe in teoria sufficiente a definire qualunque tipo di processo di generazione, tuttavia questo programma di utilità è in grado di fornire all'utente avanzato una potenza molto maggiore di quella vista fin qui.

Definizioni per bersagli multipli

Abbiamo visto come *LMK* ci permetta di definire un processo il quale, di solito, genera un prodotto ben definito come, ad esempio, il modulo eseguibile di un programma. Tale prodotto, individuato nel nostro caso da un'etichetta che lo segnala come *radice degli ascendenti*, viene anche detto «bersaglio» *[target]*. A volte, tuttavia, il prodotto finale di un processo, non è un singolo file, ma un insieme di oggetti indipendenti fra loro. Si dice allora che il file di definizione del processo è a *bersagli multipli*.

Consideriamo ad esempio un processo che, a partire da una certa base di dati, crea tre differenti rapporti (vedi figura 1). Diciamo che la base di dati contiene tutte le schede relative ad una biblioteca, e di avere scritto tre sottoprocessi che, a partire dai dati memorizzati, genera tre rapporti contenenti tutti i volumi di un autore di nome *Simak*, tutti i volumi editi dalla *Nord*, e tutti i volumi antecedenti il *1965*. I file di definizione si chiamano rispettivamente **simak.lmk**, **nord.lmk** e **ante1965.lmk**, e si trovano nel sottodirettorio **macrolmk**. Al fine di analizzare il funzionamento del file di definizione a bersagli multipli, non è necessario vedere in che modo funziona ognuno dei suddetti file di definizione. Tutto quello che è necessario sapere è che i tre file di definizione genereranno tre rapporti chiamati rispettivamente **simak.lst**, **nord.lst** e **ante1965.lst.** Inoltre, il direttorio dal quale *LMK* viene invocato, non deve contenere alcun file chiamato **fantasma**, dato che questo bersaglio non corrisponde ad un file reale, ma fa parte del meccanismo che innesca l'intero processo. Esso viene appunto chiamato «bersaglio fantasma» *[fake target]*. A questo punto non dovrebbe esservi difficile comprendere il listato riportato in figura 1.

Due note:

LMK deve far parte del percorso di ricerca comandi del processo CLI dal quale viene lanciato il comando

1>lmk -f macrolmk/rapporti.lmk

il carattere «@» usato nell'istruzione

```
#
# File "rapporti.lmk" per la generazione di tre rapporti
#

fantasma: simak.lst nord.lst ante1965.lst
   @echo "Processo terminato"

simak.lst:
   cd macrolmk
   lmk -f simak.lmk
   cd /

nord.lst:
   cd macrolmk
   lmk -f nord.lmk
   cd /

ante1965.lst:
   cd macrolmk
   lmk -f ante1965.lmk
   cd /
```

Figura 1 - File di definizione a bersagli multipli: esempio.

relativa all'ascendente fantasma, serve a specificare ad *LMK* che l'azione che segue va eseguita, ma non riportata a video. Se non fosse stato messo, il risultato del processo sarebbe stato

Lattice LMK 1.05 Copyright 1988
Lattice, Inc.
echo «Processo terminato»
Processo terminato

piuttosto che

Lattice LMK 1.05 Copyright 1988
Lattice, Inc.
Processo terminato.

Definizioni per bersagli alternativi

Se *LMK* è invocato senza specificare alcun bersaglio, il programma fa partire il processo di generazione relativo al primo ascendente che trova. È possibile tuttavia definire in uno stesso file più bersagli e decidere al tempo di invocazione quale eseguire. Consideriamo ad esempio il file riportato in figura 2.

Se eseguiamo il comando

1> lmk -f coca.lmk

il programma provvederà a copiare i tre file sul disco **df1:**, dato che il primo bersaglio (fantasma, per giunta) è **copia**. Se invece vogliamo cancellare i file suddetti, bisogna eseguire

1> lmk cancella -f coca.lmk

Regole di default

In inglese, il termine *default* vuol dire *mancante, inadempiente, in contumacia.* Nel linguaggio informatico esso indica qualcosa (un parametro, un nome) da prendere in considerazione in mancanza appunto di ulteriori specifiche. Parlare allora di *default rules*, vuol dire parlare di quelle regole che vanno considerate valide a meno che l'utente non specifichi altrimenti. Potremmo tradurre tale termine come *in difetto*, ma essendo oramai entrato nel linguaggio comune informatico, useremo il termine inglese.

Le regole di default sono quelle regole che dicono al *LMK* come trasformare un file con una certa estensione in un altro con un'etensione differente, esse si differenziano dalle azioni *[action rules]* viste in precedenza, in quanto non agiscono su uno o più file i cui nomi sono stati specificati a priori, ma su tutti i file aventi una certa estensione nell'indirizzario *[directory]*, o nel caso si stato definito un ascendente ma non la relativa azione da eseguire.

L'ordine con cui *LMK* opera sugli ascendenti e sui discendenti è il seguente, a partire dalle regole di priorità più elevata:

- azioni esplicite
- regole di default:
  - regole di trasformazione
  - regole definite dall'istruzione **.DEFAULT**
  - regole da un **.def** file
  - regole dal file **lmk.def**
  - regole interne di *LMK*

```
#
# File "coca.lmk"
#

copia:
   copy pippo to df1:
   copy topolino to df1:
   copy minnie to df1:

cancella:
   delete pippo
   delete topolino
   delete minnie
```

◀ *Figura 2 - File di definizione a bersagli alterativi: esempio.*

Figura 3 - Prove effettuate sulle regole di trasformazione. ▶

```
#
# File "prova1.lmk"
#

#
#  Copia i file con estensione .1 su quelli con estensione .2
#  $< dovrebbe contenere il nome del file da copiare completo di estensione,
#  mentre $> dovrebbe contenere lo stesso ma senza estensione.
#  Il tutto viene eseguito in RAM:
#
.1.2:
  copy $< $>.2

#
#  Bersaglio fantasma che innesca i due processi e lista la RAM:
#
prova: alfa.2 alef.2
  @list

#
#  Dovrebbe copiare alef.1 in alef.2
#
alef.2: beta.3 alef.1

#
#  Dovrebbe copiare alfa.1 in alfa.2
#
alfa.2: alfa.1 beta.3
```

Il risultato è il seguente:

```
Lattice LMK 1.05   Copyright 1988 Lattice, Inc.
        copy beta.3 beta.2
        copy alfa.1 alfa.2
RAM DISK
beta.2                          5 arwed 07-Jul-89 17:59:29
alfa.2                          5 arwed 07-Jul-89 17:59:02
prova.lmk                      96 arwed 07-Jul-89 17:58:56
alef.1                          5 arwed 07-Jul-89 17:57:53
beta.3                          5 arwed 07-Jul-89 17:21:44
alfa.1                          5 arwed 07-Jul-89 17:21:04
disk.info                     606 arwed 07-Jul-89 15:25:58
clipboards                    dir arwed 07-Jul-89 15:25:33
env                           dir arwed 07-Jul-89 15:25:33
c                             dir arwed 07-Jul-89 15:25:59
t                             dir arwed 07-Jul-89 15:25:17
6 files - 4 directories - 24 blocks used - 0 blocks free.
```

quando ci si aspettava piuttosto

```
Lattice LMK 1.05   Copyright 1988 Lattice, Inc.
        copy alef.1 alef.2
        copy alfa.1 alfa.2
RAM DISK
:
```

Vediamo in dettaglio le varie regole di default, dato che le azioni esplicite sono di fatto le istruzioni che l'utente specifica esplicitamente dopo la linea che contiene l'ascendente ed i suoi discendenti, come già visto più volte.

Regole di trasformazione

Le regole di trasformazione definiscono le azioni da intraprendere per trasformare un file con una certa estensione in un altro con estensione differente, quando non sia stata fornita alcuna azione esplicita. Ad esempio, la regola comune di trasformazione per la compilazione di un file C è la seguente.

```
.c.o:
LC $*
```

dove **$***, come già detto nella scorsa puntata, assume il nome del primo discendente comprensivo dell'eventuale percorso ma mancante dell'estensione.

Come ricorderete, altre due macro vengono definite quando si usano le regole di trasfomazione: **$<** e **$>**.Il manuale del Lattice 5.0 afferma che la prima macro viene impostata come il nome del discendente che causa l'azione (nel caso di trasformazione **.c.o**, quindi, il file di tipo **.c**), mentre la seconda prende solo il nome base dello stesso (cioè senza estensione). Alcune prove da me effettuate, tuttavia, hanno rilevato come il comportamento dell'*LMK* 1.05 (facente parte del Lattice 5.02) sia leggermente differente. In pratica non viene usato il discendente *che causa l'azione*, ma il *primo* in ogni caso. Presumo che si tratti di un baco del programma, ma potrebbe anche essere un errore del manuale. Se qualcuno ne sa qualcosa di più può contattarmi su MC-Link (MC2120). In figura 3 sono riportati i test da me effettuati.

L'istruzione .DEFAULT

Questa istruzione viene usata quando a seguito di un certo bersaglio e dei suoi discendenti, non è stata fornita alcuna azione esplicita o regola di trasformazione. Anche in questo caso il comportamento descritto nel manuale presenta delle differenze rispetto ad i test effettuati, come riportato in figura 4.

Figura 4 - Prove effettuate; sulla istruzione .DEFAULT.

```
#
# File "prova2.lmk"
#

#
#  Dovrebbe mostrare il messaggio:
#       ; Questa istruzione serve a fare alfa.3 da alfa.1 alef.1

.DEFAULT:
    ; Questa istruzione serve a fare $@ da $?

#
#  Diamo ascendente e discendenti, ma nessuna azione o
#  regola di trasformazione
#
alfa.3: alfa.1 alef.1
```

Il risultato è il seguente:

```
Lattice LMK 1.05   Copyright 1988 Lattice, Inc.
        ; Questa istruzione serve a fare $@ da
        ; Questa istruzione serve a fare $@ da
        ; Questa istruzione serve a fare alfa.3 da alfa.1 alef.1
```

quando ci si aspettava piuttosto

```
Lattice LMK 1.05   Copyright 1988 Lattice, Inc.
        ; Questa istruzione serve a fare alfa.3 da alfa.1 alef.1
```

Figura 5 - Regole interne dell'LMK per Amiga DOS.

```
#
# LMK.DEF
#
# Default rules for LMK.
#
# Licensed material.  Program property of Lattice, Inc.
# All rights reserved.
#
#

LC=LC:lc
LC1=LC:lc1
LC1FLAGS = -iINCLUDE: -iINCLUDE:lattice/
LC2=LC:lc2
AS=ASSEM:assem
LINK=BLINK:Blink

.DEFAULT:
        $(LC) $(LC1FLAGS) $*

.c.o:
        $(LC) $(LC1FLAGS) $*

.a.o:
        $(AS) $* -i Assem5.0:include -o $*.o

.h.o:
        delete $*.o
        $(LC) $(LC1FLAGS) $*
```

I file di tipo .def

Nel caso non sia stata definita neanche l'istruzione .DEFAULT, *LMK* cerca nella directory corrente un file chiamato **lmk.def.** Alternativamente l'utente può, usando l'opzione **-b**, specificare un altro file che contenga le regole da usare, anch'esso con estensione **.def.**

Le regole interne

Se neanche questi file possono essere usati, *LMK* ricorre ad un set di regole interne, differenti a seconda dell'implementazione del programma. Quelle relativa all'AmigaDOS sono riportate in figura 5.

I caratteri speciali

Abbiamo già visto che il carattere speciale «@» posto di fronte ad una azione, la esegue senza però prima mostrarla a video, come succede altrimenti. Esistono altri due caratteri speciali che, preposti ad una deteminata azione, modificano il comportamento di *LMK*.

«—» dice al *LMK* di continuare anche se l'esecuzione dell'azione in questione ha generato un qualsivoglia errore (in caso di errore, infatti, *LMK* si ferma).

« » fa da carattere di salto *[escape character]*, serve cioè quando una determinata azione inizia con un carattere che altrimenti *LMK* considererebbe speciale (ad esempio «@»).

Il carattere speciale **«&»** non è usato nella versione 1.05, se non per compatibilità con le versioni precedenti.

Nella prossima puntata vedremo le opzioni del comando *LMK* e altri bersagli fantasma. Ci sarebbe ancora molto da dire su *LMK*, ma dato che quanto

```
/*
** *** AREE ***
*/
 long           AreaDraw    (struct RastPort *, long, long);
 void           AreaEnd     (struct RastPort *);
 long           AreaMove    (struct RastPort *, long, long);
 void           InitArea    (struct AreaInfo *, short *, long);
/*
** *** RASTER TEMPORANEI ***
*/
 PLANEPTR       AllocRaster (long, long);
 void           FreeRaster  (PLANEPTR, long, long);
 struct TmpRas *InitTmpRas  (struct TmpRas *, char *, long);
```

Figura 6 - Le funzioni grafiche dell'Amiga trattate in questo articolo.

detto fin qui è più che sufficiente per usare questo programma di utilità, anche per gestire processi complessi, ho pensato di chiudere questo discorso con la prossima puntata per dare più spazio alla grafica ed in particolare introdurre la gestione della IDCMP (*Intuition Direct Communications Message Port*).

Riempimento a definizione d'area

Nella scorsa puntata abbiamo iniziato a parlare di aree e di riempimenti, ed in particolar modo abbiamo visto il *riempimento a macchia d'olio*; infatti come già detto in precedenza, la libreria grafica di Amiga mette a disposizione del programmatore due differenti tecniche di riempimento:

1. il riempimento a macchia d'olio, ed
2. il riempimento a definizione d'area.

Ricordiamo ancora una volta che entrambe queste tecniche definiscono solo la zona ed il modo con cui questa deve essere riempita. Per il colore, il modo grafico, un'eventuale mascherina bidimensionale ed il bordo esterno, se desiderato, valgono i valori validi al momento e che sono stati definiti come spiegato nella 13ª puntata.

Mentre la prima tecnica dipende dagli oggetti grafici già presenti nel *raster* su cui si sta operando, il riempimento a definizione di area si basa su una tecnica del tutto indipendente dallo stato di quest'ultimo. Di fatto essa consiste nel definire un'area chiusa, dandone i vertici. Una volta che questa è stata definita, il comando di *fine definizione* provoca il tracciamento dell'area ed il suo riempimento in accordo ai modi grafici in vigore al momento.

In figura 6 sono riportati i prototipi delle quattro funzioni grafiche che servono a definire il perimetro dell'area da riempire. Come si può vedere, oltre ad una funzione di inizializzazione e ad una di chiusura delle operazioni (rispettivamente **InitArea()** ed **AreaEnd()**), ci sono due funzioni analoghe alla **Draw()** e alla **Move()** che avevamo già visto in precedenza (cioè **AreaDraw()** ed **AreaMove()**), che servono a definire i vertici del perimetro dell'area piena.

L'indipendenza di questa tecnica consiste nel fatto che l'area è in realtà disegnata in un *raster* differente da quello su cui si sta operando, e che deve essere allocato dal programmatore prima di iniziare a definire l'area in questione. In pratica, prima di tutto il programmatore deve fare le seguenti cose:

1. allocare una struttura **AreaInfo** che serve a mantenere traccia delle operazioni che il programma farà per definire l'area da riempire;
2. allocare un vettore di parole da 16 bit sufficientemente grande da contenere le informazioni relative ai vertici del poligono che forma il perimetro dell'area (cioè un numero di byte 5 volte il numero totale di chiamate ad **AreaDraw()** ed **AreaMove()**);
3. allocare una struttura **TmpRas** per il *raster* temporaneo da usare nella costruzione dell'area;
4. allocare un *raster* temporaneo grande abbastanza da contenere l'area che si vuole tracciare.

È necessario aggiungere due parole a quanto detto, tuttavia. Innanzi tutto bisogna tener presente che l'area non è disegnata sul *raster* su cui si sta operando fintanto che non viene chiamata **AreaEnd()**. In secondo luogo, è bene allocare un vettore per i vertici con un certo margine rispetto al numero di chiamate che si intende fare ad **AreaDraw()** ed **AreaMove()**: ad esempio, se l'area da tracciare ha 7 vertici, è bene prevedere almeno nove chiamate, e cioè un vettore di 45 byte. Tale area deve essere allineata alla parola, e quindi va definita come un vettore di, nel nostro caso, 23 parole (46 byte). Per quanto riguarda invece il *raster* temporaneo, esso dipende solo dal numero di pixel contenuti di un'area rettangolare che copra interamente l'area da tracciare. Ci pensa il vostro Amiga a gestire la memoria allocata. Ecco allora che, se per una certa area è necessario un *raster* corrispondente a 100 linee per 200 colonne, tale memoria può essere riutilizzata per tracciare un'area ricopribile da un rettangolo di 40 linee per 500 colonne. Quella che conta è la dimensione totale della memoria allocata, non il rapporto tra la dimensione orizzontale e quella verticale.

In figura 6 e figura 7 sono riportate rispettivamente le funzioni e le macro utilizzate per allocare ed inizializzare il *raster* temporaneo. Figura 8 riporta invece lo scheletro di un programma che utilizza la tecnica di riempimento a macchia d'olio. Come si vede, una volta eseguiti i passi sopra elencati ed inizializzate le strutture **AreaInfo** e **TmpRas**, si può procedere a definire l'area per mezzo delle funzioni **AreaDraw()** ed **AreaMove()**. Le due funzioni di definizione dei vertici restituiscono il valore **0** se tutto è andato bene, **-1** se non c'è più posto nel vettore che mantiene le informazioni su tali chiamate. Una volta completata la definizione, chiamando la funzione **AreaEnd()**, si trasferisce l'area che nel frattempo era stata costruita dal *raster* temporaneo a quello principale. A questo punto si possono definire altre aree finché non si decide di deallocare tutte le risorse allocate.

Se si chiama **AreaMove()** dopo una sequenza di chiamate ad **AreaDraw()**, il poligono precedente viene automaticamente chiuso (come se si fosse chiamata **AreaDraw()** ancora una volta con le coordinate del primo vertice), e la penna posizionata sul primo vertice di un nuovo poligono. Lo stesso succede con **AreaEnd()**, ma in questo caso le aree definite precedentemente sono tracciate sul *raster* principale. È quindi possibile definire più aree indipendenti e chiuse prima di spostare il tutto sul piano su cui si sta operando.

```
/*
** *** RASTER TEMPORANEI ***
*/
 typedef UBYTE *PLANEPTR;

 #define RASSIZE(w,h)   ((h)*((w+15)>>3&0xFFFE))
```

Figura 7 - Le macro grafiche dell'Amiga trattate in questo articolo.

L'esercizio

Nella 14ª puntata, pubblicata sul numero 87 di MC (luglio/agosto 1989), abbiamo visto un programmino grafico detto di tipo *non interattivo*, in quanto l'utente non ha la possibilità di intervenire sulle operazioni che il programma effettua nella finestra grafica. Oggi vi propongo un altro programma, questa volta di tipo *interattivo*. Molti pensano che l'unico modo per fare della grafica interattiva sia di usare IDCMP, cioè la porta che Intuition ci mette a disposizione per dialogare con esso. In effetti questo è il metodo «classico» e quello

```
/*
**  Scheletro di codice per il riempimento a definizione di area.
**  "rp" è il puntatore al raster principale.
*/

/*
**  Alloco il raster temporaneo e la struttura TmpRas
*/
 #define LARGO  200
 #define ALTO   100
 PLANEPTR spaziolavoro;
 struct TmpRas temporaneo;

 spaziolavoro = AllocRaster(LARGO,ALTO);
 if (spaziolavoro == 0) CloseAll();
 mask |= TMPRAS;                        /* inizializzo "temporaneo"    */
 InitTmpRas(&temporaneo, spaziolavoro, RASSIZE(LARGO,ALTO));
 rp->TmpRas = &temporaneo;              /* aggancio "temporaneo" al    */
                                        /* raster principale "rp"      */
/*
**  Definiamo il vettore che conterrà le informazioni sulle chiamate
*/
 #define NP 100                         /* cioè alloco fino a 40 punti */
 WORD vettore[NP];                      /* per cinque parole per punto */
 struct AreaInfo areapiena;

 InitArea(&areapiena, &vettore[0], 2*NP/5);/* inizializzo "areapiena"  */
 rp->AreaInfo = &areapiena;             /* aggancio "areapiena" al     */
                                        /* raster principale "rp"      */
/*
** ------------------------------------------------------------------ **
**  Qui vanno le varie chiamate ad AreaMove() ed AreaDraw() per definire  **
**  l'area piena. Ad esempio:                                         **
**                                                                    **
**  error = AreaMove(rp,X0,Y0); if (error) NoArea();                  **
**  error = AreaDraw(rp,X1,Y1); if (error) NoArea();                  **
**  error = AreaDraw(rp,X2,Y2); if (error) NoArea();                  **
**                                                                    **
**  Ovviamente NoArea() è una funzione utente che va chiamata nel caso    **
**  il vettore dati sia pieno. In genere non è necessario controllare il  **
**  valore restituito da AreaDraw() ed AreaMove().                    **
** ------------------------------------------------------------------ **
*/

 AreaEnd(rp);   /* OK. Disegnamo l'area finora definita ed andiamo avanti */

/*
**  POSSIAMO CONTINUARE A DISEGNARE AREE USANDO LO STESSO RASTER TEMPORANEO
**  Quando abbiamo finito, chiudiamo tutto.
*/
 CloseAll();

 :

void CloseAll()
{
  ...
  if ((mask & TMPRAS) && spaziolavoro) FreeRaster(spaziolavoro,LARGO,ALTO);
  ...
}
```

Figura 8 - Riempimento a definizione di area.

```
MOVE 45 67      Muovi il punto corrente al punto (45,67)
DRAW 56 78      Traccia una linea dal punto corrente fino al punto (56,78)
RECT 56 89      Traccia un rettangolo dal punto corrente fino a (56,89)
FILL 90 32      Riempi a macchia d'olio a partire da (90,32)
HELP            Lista i comandi disponibili
INFO            Lista il punto corrente, il colore delle penne, ecc....
EXIT            Termina il programma
```

Tabella A

che offre la massima flessibiltià e potenza. È possibile tuttavia dialogare con il nostro programma grafico in un modo completamente differente, e cioè con una *finestra di console.*

Nelle prime puntate di questa rubrica avevamo detto che è possibile aprire una finestra di console come se fosse un normale file. Questo è vero sia per la **Open()** dell'Amiga DOS, sia per le funzioni interne del C come **open(), fopen()**, e via dicendo. Se il file di console è aperto sia in scrittura che in lettura, è possibile passare al programma dei comandi, e riceverne delle risposte. Ecco allora cosa dovete fare:

1. aprite una finestra grafica di tipo GZZ come quella descritta nella 14ª puntata (potete riusare in parte il codice di quell'esercizio);
2. aprite una finestra di console e mettetevi in attesa di comandi da parte dell'utente;
3. quando l'utente entra un comando, accertatevi che sia scritto secondo la sintassi da voi definita, altrimenti scrivete a console la sintassi corretta;
4. se l'utente lancia il comando di *fine operazioni* (EXIT, CLOSE, come volete voi), lasciate un messaggio che richieda all'utente di selezionare il gadget di chiusura e, come questo viene selezionato, chiudete tutto e terminate il programma.

Avete piena libertà nel definire il tipo di comandi e la loro sintassi, le dimensioni e le caratteristiche delle finestre. Vi consiglio comunque all'inizio di limitarvi a comandi semplici, come ad esempio quelli pubblicati nella tabella A.

Ad ogni comando può corrispondere una o più funzioni (o macro) della **graphics.library**. A questo punto non dovreste avere problemi a scrivere il programma in questione, specialmente se utilizzerete il codice usato negli esercizi precedenti.

Conclusione

Molti lettori mi hanno chiesto sempre più insistentemente di affrontare quella parte di Intuition che gestisce i menu, i *gadget*, i *requester* e via dicendo. Dato che per far questo è necessario prima spiegare alcune cose sulla IDCMP (*Intuition Direct Communications Message Port*), nella prossima puntata incominceremo a vedere come si interagisce con Intuition tramite il meccanismo a messaggi già spiegato molte puntate fa, in modo da preparare il terreno per la più impegnativa trattazione dei menu da C. Finiremo inoltre il discorso su *LMK*, e daremo una possibile soluzione dell'esercizio appena proposto. E ci sono ancora molte sorprese in arrivo...

Casella Postale

Come forse avrete notato, da un po' di tempo questa rubrica propone anche informazioni notizie e dati su vari argomenti legati in qualche modo allo sviluppo di programmi in C su Amiga (Fred Fish, libri, trucchi, ecc.). Se avete suggerimenti a riguardo potete scrivermi presso la redazione oppure mandarmi messaggi su MC-Link (MC2120).

I menu a discesa

Seguo attentamente la sua rubirca «Programmare in C su Amiga» pubblicata su MC, ed è per questo che le mando un messaggio in cui le chiedo se può darmi delle spiegazioni su come gestire i menu a discesa tramite il C (versione 4.0) e su come fare a caricare una immagine IFF. Il manuale che possiedo non affronta nessuno degli argomenti di cui lei ha parlato nei suoi articoli, ergo lei rimane l'unica mia fonte di dati. Potrebbe anche consigliarmi un buon manuale? Grazie.

Distinti saluti
Tambasco Mauro Vimodrone (Milano)

La gestione dei menu da C non è così semplice come con il Basic, ma certamente è più potente ed offre al programmatore molte possibilità, compresa quella di definire all'interno degli stessi menu elementi grafici. È tuttavia possibile rendere tale gestione molto più semplice, a condizione di limitare la flessibilità d'uso e le opzioni disponibili. Sarà proprio di queste tecniche semplificate (ma proprio per questo potenti) che parleremo in una delle prossima puntate. È tuttavia necessario introdurre un altro argomento prima, e cioè quello della porta IDCMP, senza la conoscenza della quale non è possibile introdurre menu, *gadgets*, ed altri meccanismi di interazione programma-utente messi a disposizione da Intuition. In seguito alla sua (ed ad altre richieste), ho deciso di cominciare a parlare di IDCMP nella prossima puntata, in modo da poter poi passare ai menu.

Per quello che riguarda le immagini IFF il discorso è più complesso. Scrivere un lettore generalizzato di IFF non è affatto facile. Esempi se ne possono trovare nei dischetti distribuiti dalla Commodore, di cui l'ultimo è stato incluso nel Fred Fish #185: il *Commodore IFF disk - November 1988*. Ricordo a proposito che lo standard IFF non riguarda solo le immagini (ILBM), ma i testi (TEXT), la musica (SMUS), le animazioni (ANIM), e molti altri tipi di dati ancora.

In realtà, sarebbe bene affrontare prima la gestione di file ILBM, cioè delle immagini, magari limitandosi agli inizi solo a quelle in bassa risoluzione. Ho in piano di parlare anche di IFF in seguito, ma non so ancora quando. C'è già chi mi chiede argomenti avanzati sull'AmigaDOS e sulla **layers.library** e non sarà facile accontentare i lettori più esperti senza confondere chi si è avvicinato da poco alla programmazione su Amiga, o viceversa introdurre argomenti base senza scontentare i vari *guru*.

Comunque, per chi avesse fretta, questa è una lista di programmi PD contenuti nei *Fred Fish* che leggono file IFF *[parser]* e li visualizzano sullo schermo o stampano informazioni sul tipo di file come in tabella B.

Nome	FF#	Sorgente	Descrizione
BigView	58	SI	Visualizza tutti gli ILBM (HIRES, LACE)
CheckIFF	81	NO	Verifica la struttura di un file IFF
Display	39	NO	Visualizza HAM create con un ray-tracer
Dissolve	73	SI	Come ViewILBM, ma con dissolvenza
IffLib	173	NO	Una libreria per operare su IFF
Iff2Pcs	122	SI	Genera un puzzle da un file ILBM (!!!)
Iff2Ps	94	NO	Converte file ILBM in PostScript
Iff2Sun	174	SI	Converte un file ILBM in formato Sun
ILBM2C	173	SI	Converte un file ILBM in una struct C
ILBM2Image	190	SI	Converte ILBM in un modulo C autovisual.
ShowHAM	32	NO	Visualizza HAM da CLI
View	58	SI	Visualizza ILBM da CLI o WB
ViewILBM	44	NO	Visualizza ILBM normali e HAM
Yaiffr	87	SI	Visualizza ILBM (HAM, HIRES, overscan)

Tabella B

In quanto al manuale, questa è una lista di libri editi dalla Abacus:

Amiga C for Beginners
ISBN 1-55755-045-X 280pp $19.95
Amiga C for Advanced Programmers
ISBN 1-55755-046-8 400pp $24.95
Amiga Machine Language
ISBN 1-55755-025-5 264pp $19.95
Amiga System Programmer's Guide
ISBN 1-55755-034-4 442pp $34.95

È possibile ricevere per $14.95 anche il dischetto contenente già tutti i listati riportati nei volumi in questione, con un notevole risparmio di tempo per chi volesse provare direttamente sulla macchina gli esempi descritti in questi libri, tanto più che tali sorgenti possono fornire degli ottimi scheletri per il vostro codice. L'indirizzo della Abacus è il seguente:
Abacus Dep.L3, 5370 52nd Street SE Grand Rapids, MI 49508 United States of America.

Se ordinate dall'Italia, dovete aggiungere $12 per spese postali *per ogni volume*. Quindi, se non avete la possibilità di comprare direttamente negli USA, vi consiglio di mettervi in due o tre per dividere le spese (vedi nota 2).

Note
1. Desidero scusarmi con i lettori per aver saltato un numero di MC, ma motivi personali (mi sono sposato in giugno) ed in concomitanza, necessità di carattere redazionale, han fatto sì che questa puntata uscisse con un mese di ritardo.
2. Tengo a precisare che non ho nessuno dei suddetti volumi e quindi non posso dare alcun giudizio sulla bontà degli stessi. La descrizione riportata nei cataloghi sembra comunque interessante, ed in genere il mercato americano dei libri sul software offre quasi sempre un buon livello di qualità.