Programmare in C su Amiga (21)

di Dario de Judicibus

Continuiamo lo sviluppo del nostro scheletro aggiungendo la possibilità di definire il terzo livello della struttura a menu, cioè le sottovoci, e la gestione dei comandi di selezione rapida. Alcuni nuovi automatismi completeranno il quadro delle modifiche. Inizieremo inoltre, nella scheda tecnica, la presentazione in dettaglio dei comandi dell'AmigaDOS 1.3

Nella scorsa puntata abbiamo completato lo scheletro di un programma che mostra come associare ad una finestra una struttura a menu, strutturando il codice in modo da rendere estremamente semplice effettuare eventuali modifiche, garantendo tra l'altro un'alta leggibilità dello stesso. Abbiamo anche visto inoltre come gestire i messaggi che Intuition ci spedisce a fronte delle operazioni che l'utente effettua via menu, anche qui garantendoci un elevato grado di leggibilità integrata dalla garanzia di una gestione completa ed esauriente di tutta la coda messaggi, anche in caso di selezioni multiple.

Naturalmente, essendo il programma uno scheletro, esso non contiene ancora tutte le possibili variazioni ed opzioni che un programmatore esperto può utilizzare in questo genere di programmi. Alcune di queste, come le *selezioni mutualmente esclusive*, i *sottomenu*, i *comandi*, non sono stati ancora affrontati. Il codice contiene in effetti già alcuni parametri nelle funzioni o nei campi che, in qualche modo, preparano la strada all'introduzione di tali elementi, ma, come vedremo fra poco, al momento di aggiungere la gestione di queste ulteriori opzioni, ho deciso di modificare ulteriormente le caratteristiche di alcune funzioni. D'altro canto questo scheletro è stato preparato appositamente per questa rubrica, e quindi si evolve dinamicamente con lo svilupparsi della stessa. Neppure io so quale *mostro* diventerà alla fine, ritengo tuttavia che sia il modo migliore per imparare ad usare i servizi offerti da Intuition. Alla base resta sempre il principio dei *piccoli passi*, che permette ad ogni stadio dello sviluppo di avere a disposizione un programma perfettamente eseguibile, sebbene solo una parte delle funzioni desiderate sia stata implementata.

Per motivi di spazio, come avevo già accennato nella scorsa puntata, non è più possibile riportare l'intero codice dello scheletro, per cui procederemo per *delta*. Sarebbe quindi opportuno leggere questo articolo avendo l'accortezza di tenere a portata di mano la 20ª puntata di questa rubrica. Per chi non l'avesse, consiglio caldamente di procurarsela, dato che faremo spesso riferimento ad essa. In ogni caso vedrò di caricare al più presto il codice presentato qui su MC-Link, come **SKELETON.LZH**.

skl.lmk

Il primo cambiamento che ho effettuato nel codice consiste nello scorporamento degli **#include** in un file a parte, da compilare tramite l'opzione **-ph** del compilatore. Vediamo di cosa si tratta e quali vantaggi comporta. Ricordo che

Note

1. Il programma di utilità *LMK* è stato ampiamente descritto nelle puntate che vanno dalla 14ª alla 16ª inclusa. Fate riferimento ad esse per la terminologia utilizzate in questo articolo.

2. È interessante notare come, comunque, lo scheletro che stiamo implementando, potrebbe essere adattato facilmente nel momento in cui una nuova versione di Intuition supportasse un ulteriore livello nei menu. Questo purché i nuovi elementi (sotto-sottovoci) si basino anch'essi sulla struttura **MenuItem.** Questo è uno dei vantaggi di una programmazione altamente strutturata a *scatole nere*.

```
#
# makefile for SKL
#
 NAME = Skel_003
#
 LC = lc:lc
 LINK = lc:blink
 LCOPTS = -Hskl.sym -O
 STARTUP = lib:c.o
 LIBS = lib:lc.lib+lib:amiga.lib
 LKOPTS = SD SC ND VERBOSE
 SYOPTS = -ph -oskl.sym

#
# SKL definitions
#
$(NAME): $(NAME).o
  $(LINK) FROM $(STARTUP)+$(NAME).o TO $(NAME) LIB $(LIBS) $(LKOPTS)

$(NAME).o: $(NAME).c skl.sym
  $(LC) $(LCOPTS) $(NAME).c

#
# pre-compiled table
#
skl.sym: sklhdr.c
  $(LC) $(SYOPTS) sklhdr.c

############################## DEBUG ONLY ##############################
 DNAME = skldebug
#
 LCDOPTS = -Hskl.sym -d3 -o$(DNAME).o
 LKDOPTS = ADDSYM VERBOSE
#
# SKL definitions
#
$(DNAME): $(DNAME).o
  $(LINK) FROM $(STARTUP)+$(DNAME).o TO $(DNAME) LIB $(LIBS) $(LKDOPTS)

$(DNAME).o: $(NAME).c skl.sym
  $(LC) $(LCDOPTS) $(NAME).c
```

▲ *Figura 1 skl.lmk.*

```
/*
**  File di inclusione da precompilare per generare la tabella SKL.SYM
*/
 #include "exec/types.h"
 #include "exec/memory.h"
 #include "intuition/intuition.h"
 #include "graphics/gfxmacros.h"
 #include "stdio.h"
 #include "stdlib.h"
 #include "string.h"

/*
**  Prototipi
*/
 #include "proto/exec.h"
 #include "proto/intuition.h"
 #include "proto/graphics.h"
```

Figura 2 - sklhdr.c

```
  :

/*
**  ********************************************************************  **
**  I file di inclusione sono stati scorporati e precompilati per creare  **
**  creare una tabella di simboli. Il tutto serve a rendere più veloci    **
**  le successive compilazioni.                                           **
**  ********************************************************************  **
*/

/*
**  Tipi
*/

  :

 typedef struct iDesc
 {
   UBYTE *txt;
   UBYTE  cmd;
 }
 IDESC;

/*
**  Prototipi delle funzioni interne al programma
*/

  :

 void SetupItemList( ITEM *, ITEM *, int, USHORT, ITXT *, IDESC *, ITEM ** );

  :

 #define DJ_SPEC  GIMMEZEROZERO|SMART_REFRESH|NOCAREREFRESH|ACTIVATE

  :
```

Figura 3 ▶ Primo blocco di definizioni.

quanto segue si riferisce sempre al compilatore da me usato, e precisamente il *Lattice C 5.xx*. Nel caso usiate un altro compilatore sarebbe opportuno verificare sui manuali allegati se sono disponibili le stesse funzionalità e come si attivano.

Quando si sviluppa un programma, è abbastanza comune che si effettuino molte complicazioni nel giro di poche ore, specialmente in fase di *test*, quando cioè si ritenga di aver completato il codice ad un certo livello e si desideri provare l'eseguibile per eliminare eventuali errori. È quindi opportuno adottare delle tecniche che riducano il tempo di compilazione ed ottimizzino le operazioni da effettuare per generare il modulo eseguibile. Una di queste già la conosciamo, ed è l'utilizzo del programma di utilità *LMK*. Un'altra consiste nella *precompilazione* dei file di inclusione, e nella generazione della relativa tabella dei simboli *[symbol table]*.

In figura 1 è riportato il file utilizzato per generare il programma relativo allo scheletro: **skl.lmk.** Esso è formato da due *bersagli*: il primo serve a generare il programma **Skel_003**, il secondo genera, se richiesto, un modulo che può successivamente essere utilizzato con **CodeProbe**, l'analizzatore di codice in esecuzione *[source debugger]* della *Lattice Inc.* Dato che non abbiamo ancora affrontato l'utilizzo di un analizzatore come il **CPR**, cosa che prenderebbe da sola svariate puntate di una rubrica come questa, per ora ignorate le istruzioni relative alla seconda parte di **skl.lmk**. Ho pensato di riportarle comunque per coloro che già sanno lavorare con un *source debugger*.

Concentriamoci sulle ultime due istruzioni relative al bersaglio principale *[primary target]* (vedi nota 1).

Innanzi tutto abbiamo un solo *discendente*, e ciò **sklhdr.c**. Questo file, il cui contenuto è riportato in figura 2, contiene tutte le istruzioni di inclusione che servono per la compilazione del nostro

```
/*************************************************************************
** SetupItemList: costruisce una lista di voci                          **
*************************************************************************/
void SetupItemList(ilink,ilist,itemnum,itemflags,it,itemname,slist)
  ITEM    *ilink;          /* Voce padre se sottovoci, se no NULL      */
  ITEM    *ilist;          /* Lista delle voci: vettore                */
  int     itemnum;         /* Numero di voci nella lista               */
  USHORT  itemflags;       /* Caratteristiche dell'elemento            */
  ITXT    *it;             /* Vettore di strutture IntuiText da usare  */
  IDESC   *itemname;       /* Titoli delle voci della lista e comandi  */
  ITEM    *slist[];        /* Puntatore alle liste delle sottovoci     */
{
  int i;
  SHORT   itemwidth = 0;   /* Larghezza dell'elemento                  */
  BOOL anycmd = FALSE;

  for (i = 0; i < itemnum; i++)
  {
    ilist[i].NextItem    = &ilist[i+1];  /* Lega alla voce successiva    */
    ilist[i].LeftEdge    = 0;            /* Spostamento da sinistra      */
    ilist[i].TopEdge     = HITEM * i;    /* Distanza dal bordo superiore */
    if (ilink != NULL)
    {
      ilist[i].LeftEdge += ilink->Width - WDELTA;    /* >0 per sottovoci */
      ilist[i].TopEdge  -= 2*(ilink->TopEdge/HITEM); /* negativo variab. */
    }
    ilist[i].Height      = HITEM-2;      /* Altezza dell'elemento        */
    ilist[i].Flags       = itemflags;    /* Caratteristiche della voce   */
    ilist[i].MutualExclude = 0x0000;     /* Tutti indipendenti, per ora  */
    ilist[i].Command     = itemname[i].cmd; /* Eventuali comandi         */
    if (slist != NULL)                   /* C'è una lista di sottovoci?  */
      ilist[i].SubItem   = slist[i];     /* Sì! Puntatore alle sottovoci */
    else
      ilist[i].SubItem   = NULL;         /* No! Nessuna lista.           */
    ilist[i].NextSelect  = MENUNULL;     /* Per le selezioni multiple    */

    /*
    **  Attiva e visualizza il comando "scorciatoia"
    */
    if (ilist[i].Command != '\0')
    {
      ilist[i].Flags |= COMMSEQ;
      anycmd = TRUE;             /* C'è almeno un comando nella lista    */
    }

    /*
    **  Cloniamo la struttura base ed assegnamo al campo "IText" il titolo
    **  della voce. Se è previsto un "checkmark", lasciamo suffiente spazio
    **  a sinistra.
    */
    it[i]               = basetext;
    it[i].IText         = itemname[i].txt;
    it[i].LeftEdge      += ((itemflags & CHECKIT)  ? CHECKWIDTH : 0);
    ilist[i].ItemFill   = (APTR)&it[i];

    /*
    **  Cerca la larghezza massime tra quelle delle singole voci
    */
    itemwidth  = max( itemwidth, (IntuiTextLength(&it[i]) +
     ((itemflags & CHECKIT)  ? CHECKWIDTH : 0)  ));    /* Marcatore?     */

    ilist[i].SelectFill  = NULL;         /* Testo alternativo: nullo     */
  }
  ilist[itemnum-1].NextItem = NULL;      /* Ultimo elemento              */

  /*
  **  Usa come larghezza della lista delle voci, la massima trovata.
  **  Se c'è anche un solo comando, allarga il tutto.
  */
  if (anycmd) itemwidth += 2*COMMWIDTH;
  for (i = 0; i < itemnum; i++) ilist[i].Width = itemwidth + WDELTA;
}
```

Figura 4 - SetupItemList().

programma. Dato che in genere queste istruzioni vengono modificate più raramente del resto del codice del programma, e che comunque si possono sempre includere file in eccesso senza che questo abbia un qualche effetto sulle dimensioni del modulo finale, poiché tali file *[compiler header]* contengono solamente definizioni di strutture, macro e costanti, nel caso avessimo qualche dubbio sul fatto se un certo file di include serva o meno, potremmo comunque aggiungerlo alla lista senza problemi. In effetti l'unico svantaggio sarebbe proprio nel tempo di compilazione ma, come vedremo ora, questa tecnica ha proprio lo scopo di ridurre tale tempo. Essa infatti consiste nel precompilare il file che contiene la sequenza di **#include** in modo da produrre una tabella dei simboli da utilizzare successivamente quando si compila il programma vero e proprio. L'opzione per precompilare gli *header* è appunto **-ph**, mentre **-oskl.sym** serve a salvare la tabella risultante come **skl.sym.** Questo è appunto l'ascendente relativo a questo blocco di istruzioni in **skl.lmk**.

Una volta che la tabella è stata generata, essa va referenziata nell'istruzione di compilazione del programma principale, utilizzando l'opzione **-Hskl.sym**. Tale passo ne risulta velocizzato dalle tre alle dieci volte, a seconda del numero di file di inclusione utilizzati dal programma. Ovviamente una volta generata la tabella dei simboli, non è più necessario ricompilare **sklhdr.c** ogni qual volta si modifica il codice del programma, a meno che non sorga la necessità di aggiungere un nuovo *header*. Ma di questo se ne occupa *LMK* ovviamente. Tutto quello che voi dovete fare è scrivere

lmk -f skl.lmk

ed il resto è automatico.

L'opzione **-0** serve ad attivare l'ottimizzatore globale del *Lattice C*, ma di questo parleremo in un altro momento.

A questo punto il programma eseguibile viene generato dal passo di legame *[linkage edition]*, come al solito.

Il programma scheletro

E veniamo ora al programma principale. Analizzeremo pezzo per pezzo i cambiamenti effettuati riportando solo quelle strutture o funzioni che sono state aggiunte o modificate.

Vediamo innanzi tutto la parte relativa ai tipi, alle costanti, alle strutture ed alle variabili globali. Come si può vedere in figura 3, a parte lo scorporamento dei file di inclusione, di cui abbiamo già parlato in precedenza, è stato aggiunto un nuovo tipo relativo ad una struttura formata da un puntatore ad una stringa di caratteri e ad un singolo carattere definito come **UBYTE: IDESC.** Useremo questo tipo nella definizione delle voci e delle sottovoci, più avanti. Oltre a questo nuovo tipo, è stato modificato solo il prototipo della **SetupItemList()**, essendo questa la funzione più impattata dai cambiamenti. Infine abbiamo aggiunto il valore **NOCAREREFRESH** ai segnalatori **IDCMP**, dato che, anche se abbiamo specificato in precedenza **SMART_REFRESH**, ci sono dei casi in cui Intuition manda comunque un messaggio per il restauro della finestra. Per il momento questo non sarebbe comunque potuto avvenire, dato che la finestra in questione non ha un gadget per il ridimensionamento, ma male non fa di certo...

Le modifiche maggiori riguardano tuttavia la parte relativa alla definizione della struttura dei menu. Dato che la maggior parte di tali modifiche è dovuta alla differente struttura della funzione **SetupItemList()**, analizziamo prima quest'ultima.

```
/* ================================================================ **
**              STRUTTURE DI DEFINIZIONE PER I MENU'                **
** ================================================================ */

 #define WDELTA  4

  :

/*
**  Aggiungiamo una lista di sottovoci alla terza voce del secondo menù,
**  cioè ad ITEM_230. La procedura è la stessa di quella per le voci.
**  Facciamo lo stesso per la quarta voce del primo menù.
*/
 #define SUBI_14N  2
 #define SUBI_141  0
 #define SUBI_142  1

 #define SUBI_23N  4
 #define SUBI_231  0
 #define SUBI_232  1
 #define SUBI_233  2
 #define SUBI_234  3

  :

/*
**  Dato che in genere solo alcune voci hanno una lista di sottovoci,
**  definiamo un vettore di puntatori per ciascuna voce, ma poniamoli
**  tutti nulli. Si allocheranno poi solo le strutture che servono.
**  Ci garantiamo così comunque una maggiore flessibilità, al prezzo
**  di qualche byte inutilizzato.
*/
 ITEM *subi_lis1[ITEM_1NM];      /* Vettore dei puntatori ai vettori voci */
 ITXT *subi_txt1[ITEM_1NM];      /* Vettore di tutte le strutture testi   */
 ITEM *subi_lis2[ITEM_2NM];      /* Vettore dei puntatori ai vettori voci */
 ITXT *subi_txt2[ITEM_2NM];      /* Vettore di tutte le strutture testi   */

 ITEM **subilist[] =
 {
   subi_lis1, subi_lis2, NULL
 };
 ITXT **subitext[] =
 {
   subi_txt1, subi_txt2, NULL
 };
```

▲ *Figura 5 Sottovoci: identificativi e strutture base.*

```
/*
**  Voci: caratteristiche e testi
**  NOTA: le larghezze dei menù sono state eliminate.
*/

  :

 #define ITEM_3FL  (ITEMTEXT|ITEMENABLED|HIGHBOX|CHECKIT|MENUTOGGLE)

 IDESC item_tit1[ITEM_1NM] =
 {
    {"M1: prima voce"   , '\0'}
   ,{"M1: seconda voce" , '\0'}
   ,{"M1: terza voce"   , '\0'}
   ,{"M1: quarta voce"  , '\0'}
   ,{"M1: quinta voce"  , '\0'}
 };
 IDESC item_tit2[ITEM_2NM] =
 {
    {"M2: prima voce"   , '\0'}
   ,{"M2: seconda voce" , '\0'}
   ,{"M2: terza voce"   , '\0'}
 };
 IDESC item_tit3[ITEM_3NM] =
 {
    {"M3: prima voce"   , 'A' }
   ,{"M3: seconda voce" , '\0'}
   ,{"M3: terza voce"   , 'Z' }
   ,{"M3: quarta voce"  , '\0'}
 };
 IDESC *itemname[MENU_NUM] =
 {
   item_tit1, item_tit2, item_tit3
 };
```

Figura 6 - Voci: caratteristiche e testi.

```
/*
**  Sottovoci: caratteristiche e testi
*/
 #define SUBI_14F  (ITEMTEXT|ITEMENABLED|HIGHBOX|CHECKIT|MENUTOGGLE)
 #define SUBI_23F  (ITEMTEXT|ITEMENABLED|HIGHBOX)

 IDESC subi_ti14[SUBI_14N] =
 {
    {"M14: prima sottovoce"   , '\0'}
   ,{"M14: seconda sottovoce" , '2' }
 };
 IDESC subi_ti23[SUBI_23N] =
 {
    {"M23: prima sottovoce"   , '\0'}
   ,{"M23: seconda sottovoce" , '\0'}
   ,{"M23: terza sottovoce"   , 'Q' }
   ,{"M23: quarta sottovoce"  , '\0'}
 };
 IDESC *subi_nam1[ITEM_1NM] =
 {
   NULL, NULL, NULL, subi_ti14, NULL
 };
 IDESC *subi_nam2[ITEM_2NM] =
 {
   NULL, NULL, subi_ti23
 };
 IDESC **subiname[MENU_NUM] =
 {
   subi_nam1, subi_nam2, NULL
 };
```

Figura 7 ▶ Sottovoci: caratteristiche e testi.

SetupItemList()

Innanzi tutto (vedi figura 4) la lista dei parametri è cambiata. Poiché adesso questa funzione può essere utilizzata per definire sia una lista di voci, che una lista di sottovoci, è stato aggiunto come primo parametro il puntatore alla voce a cui la lista appartiene, se di sottovoci si tratta. Ovviamente, se stiamo definendo una lista di voci, tale parametro è nullo. Abbiamo inoltre eliminato il parametro relativo alla larghezza della lista, dato che questa verrà ora calcolata automaticamente dalla stessa funzione, rendendo ancora più facile modificare il codice, lasciando allo stesso programma il compito di preoccuparsi dei dettagli. Comodo, no? Se poi volete comunque cambiare tale valore, nulla vi impedisce di farlo prima di chiamare la **SetMenuStrip()**. Il puntatore **itemname** non punta più ad un vettore di stringhe, ma ad uno di *descrittori* dell'elemento, cioè ad un vettore di tipo **IDESC**. L'ultimo parametro è ora il puntatore ad un vettore di liste di voci, piuttosto che ad una singola lista. Vedremo perché.

Veniamo al codice. Due variabili locali sono definite all'inizio. Una servirà a mantenere il valore massimo calcolato per la larghezza di ogni elemento (**itemwidth**), la seconda è una variabile logica inizializzata *falsa*. Il suo scopo è quello di mantenere memoria dell'eventuale utilizzo di un comando scorciatoia, in modo da poter successivamente aggiungere alla larghezza della lista lo spazio sufficiente alla coppia di caratteri «Amiga» «carattere» che Intuition mette in fondo al titolo della voce quando tale comando sia previsto.

La prima modifica nel ciclo principale consiste nel vedere se il primo parametro passato alla funzione sia nullo o meno. In quest'ultimo caso vuol dire che stiamo definendo una lista di sottovoci e quindi bisogna spostare a destra la lista lasciando solo una piccola sovrapposizione con la lista madre (**WDELTA**) e spostare un po' più in alto il lato superiore della lista, proporzionalmente alla posizione della voce genitrice. Ovviamente tutto ciò non è obbligatorio, questione di gusti cioè. A me piace così, quindi...

Nello scheletro presentato nella 20ª puntata, il testo relativo al singolo elemento (voce o sottovoce) era contenuto in **itemname[i]**. Ora quest'ultimo è di-

```
/*************************************************************************
** StartAll: chiamate di partenza                                       **
*************************************************************************/
void StartAll()
{
 :

 /*
 **  Voci
 */
  itemlist[MENU_100] = (ITEM *)AllocRemember(&rememory,
            ITEM_1NM * sizeof(ITEM), MEMF_CLEAR);
  if (itemlist[MENU_100] == NULL) CloseAll();
  itemlist[MENU_200] = (ITEM *)AllocRemember(&rememory,
            ITEM_2NM * sizeof(ITEM), MEMF_CLEAR);
  if (itemlist[MENU_200] == NULL) CloseAll();
  itemlist[MENU_300] = (ITEM *)AllocRemember(&rememory,
            ITEM_3NM * sizeof(ITEM), MEMF_CLEAR);
  if (itemlist[MENU_300] == NULL) CloseAll();

 /*
 **  Sottovoci
 */
  subilist[MENU_100][ITEM_110] = (ITEM *)NULL;          /* per sicurezza */
  subilist[MENU_100][ITEM_120] = (ITEM *)NULL;          /* per sicurezza */
  subilist[MENU_100][ITEM_130] = (ITEM *)NULL;          /* per sicurezza */
  subilist[MENU_100][ITEM_140] = (ITEM *)AllocRemember(&rememory,
            SUBI_14N * sizeof(ITEM), MEMF_CLEAR);
  if (subilist[MENU_100][ITEM_140] == NULL) CloseAll();
  subilist[MENU_100][ITEM_150] = (ITEM *)NULL;          /* per sicurezza */

  subilist[MENU_200][ITEM_210] = (ITEM *)NULL;          /* per sicurezza */
  subilist[MENU_200][ITEM_220] = (ITEM *)NULL;          /* per sicurezza */
  subilist[MENU_200][ITEM_230] = (ITEM *)AllocRemember(&rememory,
            SUBI_23N * sizeof(ITEM), MEMF_CLEAR);
  if (subilist[MENU_200][ITEM_230] == NULL) CloseAll();
 /*
 **  Testi (voci & sottovoci)
 */
  itemtext[MENU_100] = (ITXT *)AllocRemember(&rememory,
            ITEM_1NM * sizeof(ITXT), MEMF_CLEAR);
  if (itemtext[MENU_100] == NULL) CloseAll();
  itemtext[MENU_200] = (ITXT *)AllocRemember(&rememory,
            ITEM_2NM * sizeof(ITXT), MEMF_CLEAR);
  if (itemtext[MENU_200] == NULL) CloseAll();
  itemtext[MENU_300] = (ITXT *)AllocRemember(&rememory,
            ITEM_3NM * sizeof(ITXT), MEMF_CLEAR);
  if (itemtext[MENU_300] == NULL) CloseAll();


  subitext[MENU_100][ITEM_110] = (ITXT *)NULL;          /* per sicurezza */
  subitext[MENU_100][ITEM_120] = (ITXT *)NULL;          /* per sicurezza */
  subitext[MENU_100][ITEM_130] = (ITXT *)NULL;          /* per sicurezza */
  subitext[MENU_100][ITEM_140] = (ITXT *)AllocRemember(&rememory,
            SUBI_14N * sizeof(ITXT), MEMF_CLEAR);
  if (subitext[MENU_100][ITEM_140] == NULL) CloseAll();
  subitext[MENU_100][ITEM_150] = (ITXT *)NULL;          /* per sicurezza */

  subitext[MENU_200][ITEM_210] = (ITXT *)NULL;          /* per sicurezza */
  subitext[MENU_200][ITEM_220] = (ITXT *)NULL;          /* per sicurezza */
  subitext[MENU_200][ITEM_230] = (ITXT *)AllocRemember(&rememory,
            SUBI_23N * sizeof(ITXT), MEMF_CLEAR);
  if (subitext[MENU_200][ITEM_230] == NULL) CloseAll();
}
```

◀ *Figura 8 StartAll().*

```
/*************************************************************************
** BuildMenus: Costruisce i menù                                        **
*************************************************************************/
void BuildMenus()
{
 /*
 **  Definiamo i menù PER PRIMI
 */
  SetupMenu(&menulist[MENU_100],NULL /* nessuno */ ,
            MENU_1TX,itemlist[MENU_100]);
  SetupMenu(&menulist[MENU_200],&menulist[MENU_100],
            MENU_2TX,itemlist[MENU_200]);
  SetupMenu(&menulist[MENU_300],&menulist[MENU_200],
            MENU_3TX,itemlist[MENU_300]);

 /*
 **  Definiamo le voci PER SECONDE (attenzione all ordine, ora!)
 */
  SetupItemList(NULL,itemlist[MENU_100],ITEM_1NM,ITEM_1FL,
      itemtext[MENU_100], itemname[MENU_100],subilist[MENU_100]);
  SetupItemList(NULL,itemlist[MENU_200],ITEM_2NM,ITEM_2FL,
      itemtext[MENU_200], itemname[MENU_200],subilist[MENU_200]);
  SetupItemList(NULL,itemlist[MENU_300],ITEM_3NM,ITEM_3FL,
      itemtext[MENU_300], itemname[MENU_300],NULL              );

 /*
 **  Definiamo le sottovoci PER TERZE (attenzione all'ordine, ora!)
 */
  SetupItemList(&itemlist[MENU_100][ITEM_140],subilist[MENU_100][ITEM_140],
      SUBI_14N,SUBI_14F,subitext[MENU_100][ITEM_140],
      subiname[MENU_100][ITEM_140],NULL);
  SetupItemList(&itemlist[MENU_200][ITEM_230],subilist[MENU_200][ITEM_230],
      SUBI_23N,SUBI_23F, subitext[MENU_200][ITEM_230],
      subiname[MENU_200][ITEM_230],NULL);

 /*
 **  Fatto! E adesso associamo il tutto alla finestra.
 */
  SetMenuStrip(w,&menulist[0]);
  SaveFlags = w->IDCMPFlags;
  ModifyIDCMP(w,SaveFlags|MENUFLAGS);
  mask |= MSK_MST;
}
```

▲ *Figura 9 BuildMenus().*

ventato il puntatore ad un descrittore. Questi contiene ancora il titolo della voce o sottovoce nel sottocampo **txt**, ma in più c'è il campo **cmd** che, se non nullo, definisce il carattere alfanumerico da utilizzare in combinazione con il tasto *Amiga* per selezionare la voce da tastiera. Se tale comando è stato definito, allora è necessario aggiungere **COMMSEQ** ai segnalatori associati alla voce, se si vuole che la combinazione «Amiga» «carattere» compaia accanto al titolo della voce. Se non lo si fa, tuttavia, il comando viene attivato lo stesso, ma Intuition non mostra esplicitamente il comando accanto al testo dell'elemento.

Un altro parametro che è differente nel caso si stia definendo una lista od una sottolista, è l'ultimo. Esso è il puntatore ad un vettore di sottomenu, nel caso si stia definendo una lista di voci. Questo in quanto non esiste un livello successivo alle sottovoci (vedi nota 2). Per evitare di scrivere un'altra funzione (diciamo **SetupSubitemList()**), invece di definire una lista di sottovoci solo per quelle voci che l'hanno, ho deciso di definire un vettore di puntatori alle liste di sottovoci associate a *tutte* le voci di un certo menu. Se una voce non ha sottovoci tale puntatore sarà nullo, ovviamente, ma questo apparente spreco di pochi *byte* è controbilanciato da una maggiore flessibilità del codice qualora si intenda modificare la struttura gerarchica dei menu.

La larghezza della lista non è più definita a priori ora, ma viene calcolata utilizzando la funzione di Intuition **IntuiTextLength()**, con opportuni fattori di incremento che tengono conto delle caratteristiche dell'elemento (comandi e marcatori). Questo ha il vantaggio di tenere automaticamente conto di un eventuale *font* caricato ed utilizzato per voci e sottovoci. Alla fine del ciclo, ce n'è un altro che assegna ad ogni voce la larghezza massima trova un più piccolo incremento di sicurezza. Anche qui, questione di gusti. Da notare che, in

```
/*************************************************************************
** H_MenuPick: gestisce l'evento MENUPICK                               **
*************************************************************************/
int H_MenuPick(msg)
  IMSG *msg;
{
 :

  /*
  **  Alcune definizioni per la stampa
  */
   #define PRT_MENU(m) printf("Menù      [%s]\n",(m))
   #define PRT_ITEM(v) printf("Voce      [%s]\n",(v))
   #define PRT_SUBI(s) printf("Sottovoce [%s]\n",(s))

  /*
  **  BLOCCO PER LA GESTIONE DEI CODICI
  */
   switch (menunum)
   {
     case MENU_100:
          PRT_MENU(MENU_1TX);
          switch (itemnum)
          {
            case ITEM_110: PRT_ITEM(itemname[MENU_100][ITEM_110]); break;
            case ITEM_120: PRT_ITEM(itemname[MENU_100][ITEM_120]); break;
            case ITEM_130: PRT_ITEM(itemname[MENU_100][ITEM_130]); break;
            case ITEM_140: PRT_ITEM(itemname[MENU_100][ITEM_140]);
                 switch (subnum)
                 {
                   case  SUBI_141:
                         PRT_SUBI(subiname[MENU_100][ITEM_140][SUBI_141]);
                         break;
                   case  SUBI_142:
                         PRT_SUBI(subiname[MENU_100][ITEM_140][SUBI_142]);
                         break;
                 }
                 break;
            case ITEM_150: PRT_ITEM(itemname[MENU_100][ITEM_150]); break;
          }
          break;
     case MENU_200:
          PRT_MENU(MENU_2TX);
          switch (itemnum)
          {
            case ITEM_210: PRT_ITEM(itemname[MENU_200][ITEM_210]); break;
            case ITEM_220: PRT_ITEM(itemname[MENU_200][ITEM_220]); break;
            case ITEM_230: PRT_ITEM(itemname[MENU_200][ITEM_230]);
                 switch (subnum)
                 {
                   case  SUBI_231:
                         PRT_SUBI(subiname[MENU_200][ITEM_230][SUBI_231]);
                         break;
                   case  SUBI_232:
                         PRT_SUBI(subiname[MENU_200][ITEM_230][SUBI_232]);
                         break;
                   case  SUBI_233:
                         PRT_SUBI(subiname[MENU_200][ITEM_230][SUBI_233]);
                         break;
                   case  SUBI_234:
                         PRT_SUBI(subiname[MENU_200][ITEM_230][SUBI_234]);
                         break;
                 }
                 break;
          }
          break;
     case MENU_300:
          PRT_MENU(MENU_3TX);
          switch (itemnum)
          {
            case ITEM_310: PRT_ITEM(itemname[MENU_300][ITEM_310]); break;
            case ITEM_320: PRT_ITEM(itemname[MENU_300][ITEM_320]); break;
            case ITEM_330: PRT_ITEM(itemname[MENU_300][ITEM_330]); break;
            case ITEM_340: PRT_ITEM(itemname[MENU_300][ITEM_340]); break;
          }
          break;
   }
 :
```

Figura 10 - H_MenuPick().

caso di comandi scorciatoia, non viene aggiunta alla larghezza una quantità uguale alla costante predefinita **COMMWIDTH**, ma a due volte tale valore. Questo in quanto, a mio avviso, tale valore non è sufficiente, almeno dalle prove da me effettuate.

Da notare che non sono stati utilizzati ancora due campi: quello relativo alle selezioni mutualmente esclusive, e quello relativo al testo alternativo. Le vedremo in una versione successiva dello scheletro.

Le definizioni dei menu

Ora che abbiamo visto la nuova **SetupItemList()**, andiamo a dare un'occhiata al blocco che definisce la struttura dei menu, come promesso.

Innanzi tutto dobbiamo definire la nuova costante **WDELTA**. Quindi, per dimostrare le nuove possibilità offerte dalla funzione che definisce le voci e le sottovoci, aggiungiamo alla struttura a menu presentata nella scorsa puntata, due sottomenu: il primo associato alla quarta voce del primo menu, il secondo associato alla terza voce del secondo menu. Il terzo menu rimane senza sottovoci. Dobbiamo aggiungere allora due liste di identificativi, una per sottomenu, con le relative costanti che specificano il numero di sottovoci per sottomenu, analogamente a quanto già fatto per i menu (figura 5).

Per rendere più flessibile il codice, soprattutto in relazione all'eventuale aggiunta di un ulteriore sottomenu a quelli già definiti, definiamo una lista di puntatori per ogni menu, un puntatore per ogni voce. Essi puntano ad una lista di sottovoci, e definiscono quindi il sottomenu associato ad ogni voce. Ovviamente, se la voce non ha sottomenu, il puntatore è nullo. Analogamente definiamo una lista di puntatori ai vettori che descrivono il sottomenu per quello che riguarda il testo dell'elemento ed un eventuale comando di *selezione rapida* ad esso associato. Creiamo quindi due *sovrastrutture* **ITEM **subilist[]** e **ITEM **subitext[]** relative all'intera struttura a menu. Questo ci permetterà di indirizzare direttamente la lista di sottovoci utilizzando l'identificativo di menu più quello di voce e slegandoci così dal nome della singola lista di puntatori.

Tre cambiamenti sono stati effettuati nella parte che descrive le caratteristiche ed i testi relativi alle voci (figura 6):
1. sono state eliminate le costanti che definivano la larghezza di ogni menu, dato che ora essa è calcolata automaticamente;
2. è stato aggiunto **MENUTOGGLE** *[1.2 & 1.3]* alle caratteristiche del terzo menu, in modo che se l'utente seleziona un attributo già selezionato, esso venga deselezionato;
3. la lista di testi è stata riconvertita in una lista di descrittori, in modo da poter gestire anche i comandi di selezione rapida.

Analogamente, un blocco di istruzioni simile è stato aggiunto per i due sottomenu che stiamo definendo ora (figura 7).

A questo punto il gioco è fatto, almeno per quanto riguarda la descrizione dei vari elementi che compongono la struttura a menu associata alla finestra da aprire sullo schermo del WorkBench. Ovviamente ora dovremo fare delle modifiche anche alla procedura di inizializzazione ed a quella che crea la struttura completa. Anche qui, tuttavia, basterà solo duplicare un paio di istruzioni già esistenti e

modificarle un pochino. Man mano che andremo avanti, sarà sempre più facile aggiungere o modificare elementi nei menu, dato che il grosso del lavoro è già stato fatto, e che la maggior parte delle operazioni sono state automatizzate.

Da notare ancora una cosa, relativamente al blocco di definizioni fin qui analizzato. Mentre nella scorsa puntata abbiamo definito i vettori di stringa nel modo seguente:

```
UBYTE *item_titx[] =
{
  .
  .
  .
};
```

adesso abbiamo preferito usare per i descrittori il seguente formato:

```
IDESC *item_titx[ITEM_xNM] =
{
  .
  .
  .
};
```

LEGENDA	
<parametro>	parametro da specificare
[<opzione>]	parametro opzionale
{<opz-rip>}	parametro opzionale che può essere ripetuto n volte
...	serie che può essere continuata
|	separatore per una lista di opzioni di cui una almeno VA specificata
/A	indica che il parametro DEVE essere specificato
/K	indica che quella determinata parola chiave VA specificata se si vuole usare l'opzione ad essa associata
/S	indica una parola chiave da specificare per attivare l'operazione ad essa associata

Comando:	ADDBUFFERS
Formato:	ADDBUFFERS <unità>: <numero>
Sintassi:	ADDBUFFERS "UNITA'/A,NUMERO/A"
Scopo:	Aggiunge "cache buffers"
Specifiche:	Aggiunge un certo numero di aree di memoria alla lista dei settori utilizzati per accellerare le operazioni di I/O relative ad una certa unità (cache). Ogni area prende circa 500 byte ed un numero ottimale è di 25-30 aree. Se si usa il "file system" standard, aggiungere più di 30 aree non porta alcun vantaggio in termini di velocità. Se si usa il
Esempio:	ADDBUFFERS df1: 25 Aggiunge 25 aree al settore relativo a DF1:

Comando:	ASK
Formato:	ASK <messaggio>
Sintassi:	ASK "MESSAGGIO/A"
Scopo:	Permette di variare dall'esterno la logica di una macro comandi
Specifiche:	Emette a terminale il messaggio specificato ed, a seconda della risposta dell'utente, imposta il codice di errore in modo che, successivamente interrogato, la macro vada ad eseguire un ramo del codice piuttosto che un altro. Il codice è impostato a 5 (WARN) se la risposta è Y (si), a 0 se essa N (no) o INVIO.
Esempio:	ASK "Continuo?" IF WARN ; la risposta è SI SKIP avanti ; continua il programma (label "avanti") ELSE ; la risposta è NO echo "Ciao!" ; OK. Abbiamo finito. Salutiamo... ENDIF ; fine IF QUIT ; fine MACRO

La scheda tecnica

Con questa puntata, a grande richiesta, incomincerò a parlare dei comandi dell'*AmigaDOS 1.3*, riportando una descrizione di tutti quei comandi che sono stati in qualche modo modificati rispetto la versione precedente. Ovviamente saranno necessarie alcune puntate per riportare la lista completa. Per non fare favoritismi andrò in ordine strettamente alfabetico...

Comando:	ASSIGN
Formato:	ASSIGN [[<nome> <direttorio>] [LIST] [EXIST] [REMOVE]
Sintassi:	ASSIGN "NOME,DIRETTORIO,LIST/S,EXISTS/S,REMOVE/S"
Scopo:	Assegna un nome logico ad un direttorio
Specifiche:	Assegna un nome logico ad un direttorio * oppure * se solo il nome è specificato, lo rimuove dalla lista * oppure * se solo il comando è specificato, visualizza la lista corrente * oppure * se LIST è specificato, rimuove od assegna il nome, a seconda se è presente solo il primo od entrambi i primi due parametri, quindi visualizza la lista risultante * oppure * se EXISTS è specificato, cerca il nome nella lista: se lo trova lo visualizza nella finestra CLI, altrimenti imposta il codice di errore a 5 (WARN). * oppure * se REMOVE è specificato, rimuove il nome dalla lista ma NON libera le risorse associate - questa opzione è stata introdotta ESCLUSIVAMENTE per prove di sviluppo, e può causare un GURU!
Esempio:	ASSIGN fonts: df2:fnt Assegna il nome logico "FONTS:" a "df2:fnt"

Comando:	AVAIL
Formato:	AVAIL [CHIP|FAST|TOTAL]
Sintassi:	AVAIL "CHIP/S,FAST/S,TOTAL/S"
Scopo:	Visualizza l'utilizzo della memoria
Specifiche:	Visualizza la memoria disponibile, quella in uso, la massima presente nel sistema, e il blocco più largo di memoria contigua disponibile al momento, in byte.
Esempio:	AVAIL Visualizza le informazioni suddette per tutti i tipi di memoria RISULTATO: Type Available In-Use Maximum Largest chip 312728 210504 523232 303384 fast 497192 543984 1041176 433552 total 809920 754488 1564408 433552

Da un punto di vista pratico, i due formati sono equivalenti, almeno nel nostro caso. Il secondo presenta tuttavia un vantaggio. Supponiamo che la nostra lista sia formata da sei elementi, e che noi, per errore, definiamo sette stringhe, cioè una di più. Nel primo caso il compilatore accetta tale definizione, col risultato di sprecare un certo numero di byte, nel secondo caso ci dà una segnalazione di errore. Se il numero di stringhe è invece inferiore, la compilazione va a buon fine in entrambi i casi, almeno con il *Lattice C*. Io personalmente avrei preferito essere avvertito in quest'ultimo caso, a condizione di aver usato il secondo formato, di ricevere cioè un *warning*, in modo da evitare di dover eseguire il programma per accorgermi di aver dimenticato un elemento. Per la terza volta... questione di gusti. Sta di fatto che personalmente trovo il secondo formato più chiaro, avendo dovuto comunque definire le costanti **ITEM_xNM** per altri motivi.

StartAll()

Questa, come certamente ricorderete, è la procedura di inizializzazione, cioè quella *routine* che apre le librerie e la finestra, ed alloca tutte le strutture che ci servono utilizzando il potente servizio di Intuition **AllocRemember()**.

Due cambiamenti sono stati effettuati in questa procedura (vedi figura 8).

Il primo è relativo alle istruzioni che allocano memoria per le voci e relative strutture: in pratica si sono sostituiti gli identificativi di menu ad i numeri che si erano esplicitamente codificati nella versione precedente dello scheletro.

Il secondo consiste nell'aggiunta delle strutture che, analogamente a quanto già fatto con le voci, allocano memoria per le sottovoci e strutture relative. Da notare come, a tali linee di codice, sono state aggiunte una serie di assegnazioni a **NULL** di quei puntatori che corrispondono a voci che non hanno alcun sottomenu associato. In effetti questo non era strettamente necessario. Ha comunque due vantaggi. Innanzitutto ci pone al riparo da errori o comportamenti anomali del compilatore, che, anche se dovrebbe *sempre* inizializzare a zero tutte le variabili interne, potrebbe non farlo o perché chi l'ha scritto non si è attenuto ad una delle principali raccomandazioni *ANSI*, o per un semplice baco nel codice. Mettersi al riparo non costa nulla. Il secondo vantaggio consiste nel preparare uno schema che permette di dare una maggiore leggibilità al codice evidenziando *esplicitamente* quali voci hanno un sottomenu associato e quali no, e di preparare così, di fatto, la strada all'aggiunta di nuovi sottomenu.

Anche in questo caso, molte delle istruzioni utilizzate nel nostro programma, se non addirittura lo stile stesso di programmazione, ha come scopo principale, non tanto quello di ottimizzare le operazioni o la logica interna del programma stesso, quanto quella di permettere al programmatore di rimettere le mani sul programma senza doversi preoccupare di controllare ogni volta la consistenza del codice, per ogni minimo cambiamento. Ad esempio, se dovete definire una lista di, diciamo, otto menu, con sette voci in media per menu, ed una cinquantina di sottovoci sparse qua e là in varie sottoliste, il concentrare le operazioni relative alla struttura nel suo complesso in pochi blocchi ben definiti diciamo, uno per l'allocazione di memoria, uno per l'inizializzazione, uno per la deallocazione — e l'ordinare ogni singolo blocco seguendo un qualche criterio di ordinamento, permette di identificare immediatamente eventuali discrepanze e disallineamenti nella definizione della struttura stessa. Eviteremo così, ad esempio, di definire due volte la stessa voce, o di duplicare un attributo in due menu differenti, od ancora di «dimenticarci» di definire un elemento. D'altra parte, se avessimo scritto un codice più criptico, non avremmo poi salvato molti byte, ma avremmo sicuramente perso in chiarezza ed avremmo aumentato le possibilità di errore in fase di edizione del codice stesso. Tanto per fare un esempio, lo scheletro qui presentato, generato tramite *LMK* con le definizioni in figura 1 (bersaglio principale), viene ad essere grande alla fine solo 9408 byte, a fronte di un sorgente di ben 30432 byte.

BuildMenus()

Un'altra procedura coinvolta nei cambiamenti dovuti all'introduzione dei sottomenu è la **BuildMenus()** (figura 9).

Anche qui i cambiamenti sono di due tipi.

Il primo riguarda un più coerente utilizzo degli identificativi di menu e di voce eliminando definitivamente l'uso di valori espliciti per tali campi. Ecco allora che, ad esempio, **MENU_100** va a sostituire il valore 0 nella prima istruzione della procedura.

Il secondo è dovuto alla differente sintassi della funzione **SetupItemList()** da una parte, ed, ovviamente, all'aggiunta delle chiamate per la definizione dei due nuovi sottomenu, dall'altra. Da notare l'utilizzo costante e continuo delle costanti che rappresentano gli identificativi dei menu, delle voci e delle sottovoci. Tale utilizzo è tanto più vantaggioso quanto più si usano identificativi *parlanti*, come ad esempio:

```
:
#define MENU_EDIT  2
#define MENU_PRNT  3
:
#define EDIT_COPY  1
#define EDIT_CUT   2
#define EDIT_PASTE 3
:
```

H_MenuPick()

Per finire, vediamo come è cambiata la *routine* di gestione degli eventi. Anche in questo caso abbiamo due modifiche, una sempre relativa all'aggiunta dei sottomenu, come si può vedere in figura 10, l'altra dovuta all'introduzione di tre macro per la visualizzazione a terminale degli elementi selezionati. Tali macro permettono di modificare facilmente il formato di stampa senza dover mettere mano alle decine di linee di codice presenti nel *blocco per la gestione dei codici*. In futuro esse potrebbero essere utilizzate anche per chiamare una serie di procedure interne in grado di gestire in modo più complesso le informazioni passate da Intuition.

Anche in questa procedura abbiamo introdotto l'uso degli identificativi al posto degli indici espliciti.

Conclusione

Ed anche per questa volta è tutto. Nella prossima puntata vedremo quei campi della struttura **MenuItem** che ancora non abbiamo approfondito. Vedremo inoltre come lo scheletro fin qui proposto non impedisce affatto la gestione di strutture a menu più complesse, con elementi grafici o stili originali e non ortodossi. Nel frattempo, provate a scrivere qualche programma vostro basato sullo scheletro che fin qui abbiamo impostato, od a riconvertire qualche vostro vecchio programma. In quest'ultimo caso potrebbe essere interessante comparare i due programmi per vedere se c'è stato un aumento in byte del modulo eseguibile, e se sì, se questo incremento è rapportabile ai benefici derivanti della maggiore flessibilità e leggibilità della versione basato sullo scheletro.

Che dite, ne abbiamo fatta di strada da quando uscì la prima puntata di questa rubrica, nel lontano maggio del 1988, no?