Programmare in C su Amiga (22)

di Dario de Judicibus

Le dimensioni che il nostro scheletro sta assumendo, sono tali da richiedere una ridefinizione della struttura del programma. Ne approfitteremo per ribadire alcuni concetti importanti relativi allo sviluppo di grossi programmi. Il nuovo scheletro implementerà in più i testi alternativi, finora non utilizzati, e proporrà nuove ed interessanti sfide al programmatore

Chi mi ha seguito nelle ultime quatro o cinque puntate di questa rubrica, è probabile che, cercando di riprodurre il programma scheletro sul suo Amiga, si sia reso conto come questo si sia evoluto non solo in termini di codice, ma anche di struttura. Nella scorsa puntata, inoltre, abbiamo introdotto un file di tipo **.lmk** per meglio gestire la tecnica degli **headers** precompilati da cui si ricavano le tabelle simboliche da utilizzare in fase di compilazione. Il tutto era stato fatto con lo scopo di velocizzare i tempi di compilazione. Durante un processo di sviluppo, infatti, specialmente se si adotta la tecnica a piccoli passi che permette di andare avanti raggruppando le modifiche e le aggiunte al programma in blocchi logicamente connessi e generando di volta in volta moduli che possono essere seguiti pur non avendo ancora completato il programma (se mancano cioè ancora un certo numero di funzioni), ci si ritrova spesso a compilare molte volte lo stesso programma nel giro di poche ore. Non solo. Quando la compilazione produce degli errori, od addirittura l'eseguibile perfettamente compilato si comporta in modo diverso da quanto previsto e magari manda in GURU il nostro Amiga, il numero di compilazioni all'ora sale vertiginosamente. Classico è il caso in cui una compilazione dia una sfilza di errori. Si procede allora ad analizzare il primo, si identifica il problema nel codice sorgente, lo si corregge e si ricompila il tutto. In genere a questo punto possono succedere tre cose:

1. il messaggio di errore relativo a quello specifico problema scompare;
2. sia il messaggio di cui sopra, sia una buona parte dei messaggi successivi scompare (vedi nota 1);
3. il primo messaggio scompare, ma in compenso ne compaiono molti altri (vedi nota 2). Non preoccupatevi. Spesso basta eliminare quello che ora è il nuovo primo messaggio per cancellare tutti gli altri.

Negli ultimi due casi non bisogna farsi prendere dal panico. Si tratta di situazioni che si verificano spesso quando si sviluppa un programma e che sono dovute al fatto che le capacità del compilatore di interpretare un sorgente sono comunque limitate e spesso affette da assunzioni che lo stesso compilatore cerca di fare a fronte di una istruzione non corretta.

Due casi classici che in C prendono spesso in contropiede anche chi sviluppa da anni in questo linguaggio sono l'utilizzo dell'operatore *uguale* al posto del *doppio uguale* nelle condizioni di uguaglianza, ed il dimenticarsi un punto e virgola in fondo ad una istruzione o metterne uno di troppo dopo una istruzione #**define** o tra una istruzione ed il

Errata Corrige

Nella scorsa puntata, nel codice della procedura **H_MenuPick()** riportato in figura 10, c'è una imprecisione che non permette di visualizzare i testi base delle voci e sottovoci durante l'esecuzione del programma. In pratica basta modificare tutte le istruzioni del tipo:

```
case ITEM_XY0: PRT_ITEM(itemname[MENU_X00][ITEM_XY0]); break;
...
case SUBI_XYZ: PRT_SUBI(subiname[MENU_X00][ITEM_XY0][SUBI_XYZ]);
```

nel modo seguente:

```
case ITEM_XY0: PRT_ITEM(itemname[MENU_X00][ITEM_XY0].txt); break;
...
case SUBI_XYZ: PRT_SUBI(subiname[MENU_X00][ITEM_XY0][SUBI_XYZ].txt);
```

Si è trattato di una svista dovuta al fatto che ho riportato in figura una versione precedente della procedura che funzionava prima della definizione della struttura iDesc.

successivo blocco racchiuso tra parentesi graffe.

Dato che i tempi di compilazione crescono con le dimensioni del programma, si capisce facilmente come non convenga lasciare che un sorgente cresca troppo in lunghezza. Come ho già avuto modo di dire quando abbiamo parlato di *LMK*, la soluzione consiste nello spezzare il programma in più file. Ma quali criteri logici vanno utilizzati nel fare ciò? Se da un estremo c'è il singolo file per tutto il programma, dall'altro c'è la suddivisione in un file per procedura. Vediamo quali sono i vantaggi e gli svantaggi di quest'ultima, cioè nel caso che ogni procedura sia mantenuta in un solo file.

I vantaggi

- Una maggiore facilità nell'individuare gli errori nel codice, dato che ora ogni gruppo di messaggi di errore si riferisce ad una singola procedura, riducendo inoltre la possibilità di incappare nell'effetto schermo (vedi nota 2).
- Una notevole riduzione dei tempi di ricompilazione a fronte della correzione dei vari errori trovati, dato che basta ricompilare solo il file contenente la procedura interessata dalle modifiche, e riassemblare il tutto con il comando di **link**.

Gli svantaggi

- Una minore visibilità del programma nel suo complesso ed una maggiore difficoltà in fase di edizione del codice stesso, specialmente quando sorge l'esigenza di effettuare una modifica che interessa più file contemporaneamente. Un esempio sono le modifiche alle interfacce tra moduli.
- Una maggiore probabilità di introdurre errori a fronte di modifiche del codice, specialmente se queste non interessano procedure del tipo *a scatola nera*, e quindi richiedano un fasamento del codice sotto modifica con blocchi di istruzioni situati in altre procedure.
- Un incremento dei tempi di compilazione a fronte di modifiche che hanno interessato molte procedure, a causa del maggior numero di chiamate al programma di compilazione ed all'aumentato numero di operazioni di I/O effettuate dal programma di assemblaggio o *linkage editor*. Questo problema può essere notevolmente ridotto qualora si utilizzi un programma *cache* tipo **FACC II**.

Come capita spesso, quindi, la soluzione ideale sta nel mezzo. In pratica, consiste nel raggruppare le varie procedure in vari file, secondo criteri che dipendono generalmente dal tipo di programma che si sta sviluppando. Ad esempio, se un certo gruppo di funzioni elementari è candidato a diventare una libreria condivisibile *[shared library]*, come appunto quelle di sistema, viene immediato raggruppare tutte le procedure in un singolo file. In seguito si potrà eliminare il file in questione, riconvertendolo nel sorgente di una libreria, ed aggiungere al programma l'opportuna istruzione di OpenLibrary() relativa alla nuova libreria.

Il programma scheletro

Vediamo quale criterio abbiamo adottato in questa puntata per il programa scheletro.

Il codice del programma vero e proprio è stato suddiviso in due parti. Da una parte ci sono quelle funzioni che sono candidate a diventare di tipo generale, slegate cioè dal programma in questione e riutilizzabili così come sono, in altri programmi. Un esempio è la **CloseSafelyWindow()**. Queste procedure, che chiameremo d'ora in poi *funzioni di servizio*, dovranno alla fine avere le seguenti caratteristiche:

- essere del tutto indipendenti dalla logica del programma che le chiama;
- essere disegnate a *scatola nera*, non utilizzare cioè alcuna variabile globale ma solo i parametri passati dal programma chiamante;
- se utilizzano costanti, macro, o tipi utilizzati anche da altre funzioni o dal programma principale, queste devono essere raggruppate in uno o più file di inclusione.

Questa suddivisione ha il vantaggio di ridurre la ricompilazione quasi esclusivamente al file che contiene il programma

Note

1. Si chiama *effetto valanga* quella situazione per cui un errore in un certo punto del codice, oltre a generare un messaggio relativo a quel determinato errore, confonde a tal punto il compilatore da impedirgli di interpretare correttamente buona parte delle istruzioni successive. Questo provoca tutta una serie di messaggi di errore che non si riferiscono a bachi veri e propri, ma che son solo la conseguenza del primo errore. Essi infatti spesso scompaiono del tutto quando si corregge l'errore che aveva scatenato l'effetto valanga. Questo è uno dei motivi per cui, quando vengono generati molti messaggi di errore, è più pratico affrontarli uno alla volta, ricompilando il tutto dopo ogni modifica, piuttosto che cercare di identificarli tutti. Il programmatore esperto, tuttavia, spesso acquisisce una tale sensibilità nel riconoscere i messaggi relativi ad errori veri e propri da quelli generati dall'effetto valanga, da identificare la maggior parte degli errori reali e ridurre così il numero di compilazioni necessarie a cancellare tutti i messaggi prodotti dal compilatore. A riguardo, raccomando di fare uno sforzo per eliminare sempre *tutti* i messaggi, anche quelli di avvertimento *[warning]*, anche se in linea di massima essi non fanno riferimento a problemi tali da impedire il funzionamento del programma.

2. Si chiama *effetto schermo* quella situazione per cui un errore nel programma scherma un certo numero di errori nelle istruzioni successive, facendo credere al compilatore che esse siano corrette. In questo caso, nel momento in cui il primo viene risolto, il compilatore si accorge di tutti o parte dei successivi, generando così molti messaggi di errore là dove prima ce n'era solo uno. Quando tale effetto si somma, per uno o più degli errori precedentemente schermati, a quella valanga, il risultato è tale da far prendere un colpo ai programmatori meno esperti, e qualche volta a far bestemmiare anche quelli più esperti, specialmente se è già qualche ora che si sta lavorando per mettere a punto il programma. Un fattore psicologico importante, con un risvolto anche pratico, è quello di mantenere sempre copia dell'ultimo sorgente che non aveva dato errori di compilazione, prima cioè delle ultime modifiche. Tale tecnica dà al programmatore la garanzia di essere sempre in grado di ritornare ad un livello dello sviluppo da cui eventualmente ripartire, se l'introduzione di nuove modifiche ha talmente sconvolto il codice da rendere più semplice ricominciare da zero piuttosto che intestardirsi sulla matassa di istruzioni formatasi. Questa tecnica si chiama a *soglia [checkpoint]*, e ben si sposa con lo sviluppo a piccoli passi.

3. Si dice che un oggetto è consolidato quando esso ha raggiunto la struttura finale come da progetto, per cui eventuali successive modifiche sono dovute alla scoperta di errori nella implementazione di tale oggetto oppure a variazioni del progetto originale.

4. D'ora in poi chiameremo *fase di assemblaggio* quella che in inglese è chiamata *linkage edition*. Fate attenzione quindi a non confonderla con la fase di compilazione di sorgente scritto in linguaggi *Assembler*. La scelta di tale termine è dovuta al fatto che la traduzione letterale del termine inglese *linkage*, e cioè legame, legatura, rilegatura, non suona molto bene in italiano.

```
#
# makefile for SKL
#
 NAME = skeleton
 MAIN = sklmain
 PROCS = sklprocs
 STDH = sklstdh
 USRH = sklusrh
#
 LC = lc:lc
 LINK = lc:blink
 LCOPTS = -H$(STDH).sym -H$(USRH).sym -O
 STARTUP = lib:c.o
 LIBS = lib:lc.lib+lib:amiga.lib
 LKOPTS = SD SC ND VERBOSE

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

$(MAIN).o: $(MAIN).c $(STDH).sym $(USRH).sym
  $(LC) $(LCOPTS) $(MAIN).c

$(PROCS).o: $(PROCS).c $(STDH).sym $(USRH).sym
  $(LC) $(LCOPTS) $(PROCS).c

#
# pre-compiled tables
#
$(STDH).sym: $(STDH).c
  $(LC) -ph -o$(STDH).sym $(STDH).c

$(USRH).sym: $(USRH).c $(STDH).sym
  $(LC) -ph -H$(STDH).sym -o$(USRH).sym $(USRH).c
```

Figura 1 - skl.lmk.

```
/*
**  File di inclusione da precompilare per generare la tabella SKLSTDH.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 - sklstdh.c.

```
/*
**  Definizioni da precompilare per generare la tabella SKLUSRH.SYM
*/

/*
**  Tipi
*/
 typedef struct Node         NODE;
 typedef struct Message      EMSG;
 typedef struct IntuiMessage IMSG;
 typedef struct IntuiText    ITXT;
 typedef struct Menu         MENU;
 typedef struct MenuItem     ITEM;

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

Figura 3 - sklusrh.c.

vero e proprio, dato che le funzioni di servizio hanno un livello di stabilità nel processo di sviluppo molto maggiore. Una volta consolidate, inoltre, a meno di errori nel codice, non devono più essere ricompilate, anche se il programma principale cambia pesantemente. Qualora poi nasca l'esigenza di sviluppare nuove funzioni di servizio, quest'ultime potranno essere mantenute in un terzo file fino a consolidamento avvenuto (vedi nota 3), per poi raggruppare i due file in una singola unità di compilazione contenente tutte le funzioni di servizio. Se tali funzioni, poi, sono sufficientemente elementari e logicamente connesse — ad esempio un insieme di procedure matematiche per il calcolo delle funzioni di Bessel, di Laplace, di xxxx, e via dicendo, oppure un set di funzioni per costruire *plot* di funzioni o dati sperimentali — può valere la pena di assemblare in una libreria di compilazione o di una *run-time* a seconda del tipo di funzioni e del loro utilizzo.

Vediamo ora la nuova struttura che abbiamo dato al nostro scheletro. In tutto ora abbiamo quattro file sorgente più quello di specifiche per *LMK:* Vediamo cosa contengono:

Attributes	Date	Time	Blocks	Bytes	Name
-----rwed	90-02-19	18:52:26	2	725	skl.lmk
-----rwed	90-02-19	19:16:30	48	23041	sklmain.c
-----rwed	90-02-19	20:07:50	22	10507	sklprocs.c
-----rwed	90-02-19	18:48:44	1	366	sklstdh.c
-----rwed	90-02-19	19:17:11	1	383	sklusrh.c

sklmain.c È il file principale, quello cioè che contiene il **main().** Contiene tutte le procedure specifiche del programma, le definizioni dei menu, e le variabili globali. In particolare contiene le seguenti funzioni:

- main()
- StartAll()
- CloseAll()
- BuildMenus()
- LetsGo()
- HandleEvent
- H_MenuPick()
- H_CloseWindow()

e i seguenti tronconi:

- H_MenuVerify()
- H_MouseButtons()

sklprocs.c Contiene tutte le procedure generali, che cioè potranno alla fine essere riutilizzate anche in altri programmi. Tra parentesi, una volta consolidate, non sarà necessario ricompilare il file ogniqualvolta vorremo introdurle in un nuovo programma, ma basterà riassemblare il programma con il file **sklprocs.o** generato in precedenza. Vediamo quali sono per ora queste funzioni:

- SetupMenu()
- SetupItemList()
- CloseSafelyWindow()

sklstdh.c Questo è il file che contiene tutti gli include da precompilare e che nella scorsa puntata avevamo chiamato **sklhdr.c**. Esso serve a generare la tabella simbolica contenente tutte le costanti, le macro ed i tipi (**typedef** e stutture) forniti dal sistema (eg. Intuition, EXEC, AmigaDOS, e via dicendo). sklusrh.c È l'equivalente di **sklstdh.c** per le costanti, le macro ed i tipi definiti dal programmatore. Il fatto di mantenere le

due tabelle simboliche separate è dovuto al fatto che, mentre la prima è di fatto consolidata (al più potremmo decidere di aggiungere un nuovo file di inclusione, a fronte di nuove funzionalità nel programma), la seconda è continuamente soggetta a modifiche, evolvendosi con lo sviluppo del programma principale.

Come si può vedere in figura 1, analizzando cioè il file di specifiche per *LMK*, la fase di compilazione genera due moduli oggetto (**sklmain.o** e **sklprocs.o**) e due tabelle simboliche (**sklstdh.sym** e **sklusrh.sym**) utilizzate nella generazione dei moduli oggetto stessi (vedi figura 2 e figura 3). La fase di assemblaggio finale utilizza come sempre anche il modulo di avvio **c.o**, generando il programma **skeleton**. Da notare l'utilizzo delle prime cinque costanti per i nomi base dei file (quelli cioè senza estensione).

La suddivisione del programma in quattro file sorgente ha tuttavia richiesto qualche cambiamento anche al codice dei singoli file, per evitare problemi in fase di assemblaggio dei moduli oggetto (vedi nota 4).

Innanzi tutto abbiamo separato i prototipi delle funzioni riportate in testa al file **sklmain.c** in due blocchi: il primo si riferisce alle funzioni esportate nel file **sklprocs.c** e denominate d'ora in poi *esterne* (la parola chiave **extern** è sottintesa di fronte ad ogni prototipo); il secondo riporta i prototipi delle funzioni presenti più avanti nello stesso file, e d'ora in poi dette *interne* (vedi figura 4).

Poi, come già detto, abbiamo scorporato i tipi definiti dal programmatore creando così il file **sklusrh.c** per generare la tabella simbolica utente (vedi figura 3).

Quindi abbiamo estratto dal precedente scheletro le costanti usate da **SetupMenu()** e **SetupItemList()** e la struttura prototipo di tipo **IntuiText** e le abbiamo messe in testa al file **sklprocs.c**, come si può vedere in figura 5.

Nella stessa figura compaiono anche due riferimenti esterni. In futuro dovremmo cercare di eliminarli. Essi infatti evidenziano come le due procedure **SetupItemList()** e **CloseSafelyWindow()** non siano ancora completamente a scatola nera come dovrebbero essere, ma dipendono appunto ancora da due variabili globali. Nella prossima puntata vedremo di risolvere questo problema e di rendere così più pulita l'interfaccia delle funzioni di servizio.

```
/*
**  Prototipi delle funzioni esterne al programma
*/
 void SetupMenu     ( MENU *, MENU *, BYTE *, ITEM * );
 void SetupItemList( ITEM *, ITEM *, int, USHORT, ITXT *, IDESC *, ITEM ** );
 void CloseSafelyWindow ( struct Window *, struct TextFont * );

/*
**  Prototipi delle funzioni interne al programma
*/
 void StartAll      ( void );
 void CloseAll      ( void );
 void BuildMenus    ( void );
 void LetsGo        ( void );
 int  HandleEvent   ( IMSG * );
/* ------------------------------------------------------------ STUBs --- */
 int  H_MenuVerify    ( IMSG * );
 int  H_MouseButtons  ( IMSG * );
/* ------------------------------------------------------------------- */
 int  H_MenuPick      ( IMSG * );
 int  H_CloseWindow     ( IMSG * );
```

Figura 4 - Variazioni in sklmain.c.

```
/*
**  Riferimenti esterni
*/
 extern struct MsgPort *up;
 extern struct Remember *rememory;

/*
**  Costanti
*/
 #define HITEM  10
 #define WDELTA  4
 #define CHARWIDTH 10

/*
**  Strutture
*/
 ITXT basetext =        /* Questa struttura fa da base ai testi delle voci */
 {
   0, 1,                /* Penne per il tratto e per lo sfondo             */
   JAM2, 5,             /* Modo grafico e spostamento dal bordo sinistro   */
   0,                   /* Spostamento dal bordo superiore                 */
   NULL,                /* Font usato, se diverso da quello di sistema.    */
   NULL,                /* Testo (da riempire)                             */
   NULL                 /* Eventuale struttura successiva                  */
 };
```

Figura 5 - Variazioni in sklprocs.c.

```
/*
**  Voci: caratteristiche e testi
*/

  :

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

  :

/*
**  Sottovoci: caratteristiche e testi
*/

  :

 IDESC subi_ti14[SUBI_14N] =
 {
    {'\0', "M14: prima sottovoce"   , NULL}
   ,{ '2', "M14: seconda sottovoce" , NULL}
 };
 IDESC subi_ti23[SUBI_23N] =
 {
    {'\0', "M23: prima sottovoce ON", "M23: prima sottovoce OFF"}
   ,{'\0', "M23: seconda sottovoce" , NULL}
   ,{ 'Q', "M23: terza sottovoce"   , NULL}
   ,{'\0', "M23: quarta sottovoce"  , NULL}
 };
```

Figura 6 - Descrittori delle voci e delle sottovoci.

I testi alternativi

Vediamo ora quali cambiamenti sono stati apportati al codice vero e propri e quali nuove possibilità abbiamo introdotto.

Nella scorsa puntata avevamo visto come associare a voci e sottovoci un comando per la selezione rapida, ma non avevamo ancora implementato la possibilità di avere un testo alternativo a quello base associato all'elemento in questione. La **SetvpItemList()** si limitava infatti ad impostare il campo **SelectFill** a NULL. Vediamo adesso come attivare tale campo.

Per prima cosa è necessario modificare la struttura associata al tipo **IDESC** come si può vedere in figura 5, aggiungendo il puntatore alla stringa che contiene il testo alternativo e, per comodità, spostando il carattere corrispondente al comando di selezione rapida in testa alla struttura. Di conseguenza i blocchi di codice relativi a questa struttura vengono modificati come riportato in figura 6.

Come si può vedere in figura, solo due voci ed una sottovoce hanno il testo alternativo non nullo. Questo è stato fatto di proposito per evidenziare il seguente problema:

Dato che per ogni testo base od alternativo che sia, è necessario allocare una struttura **IntuiText**, *l'implementazione dei testi secondari avrebbe conseguenza, se utilizzassimo la tecnica già adottata per i testi base, di raddoppiare la quantità di memoria allocata per i testi delle voci e delle sottovoci. Tuttavia, mentre voci e sottovoci hanno sempre e comunque un testo base (a meno che non contengano elementi grafici), il numero di testi secondari è spesso di molto inferiore a quello dei testi base. Ne risulta quindi uno spreco di memoria allocata e non utilizzata.*

Come fare allora a gestire i testi secondari e ad evitare contemporaneamente questo spreco di memoria?

Una prima soluzione è visibile in figura 7 (vedi ultima pagina) che riporta la nuova **SetupItemList()**. Vediamo i cambiamenti effettuati rispetto alla volta scorsa.

In testa alla procedura abbiamo aggiunto un contatore che terrà il conto di quanti testi alternativi abbiamo effettivamente, ed un puntatore alla lista delle strutture **IntuiText** che conterranno tali testi. Tale puntatore **altname** è analogo a quello già usato per i testi base (**itemname**) ma, a differenza di quest'ultimo, è una variabile locale, non globale. Viene fuori così già una prima asimmetria. Vedremo che essa non è assolutamente necessaria, ma che è semplicemente la conseguenza del tipo di soluzione scelta.

All'interno del ciclo principale che riempie le varie strutture **MenuItem**, abbiamo per ora lasciato l'assegnazione a NULL del campo **SelectFill**. È stato però aggiunto il conteggio del numero di testi alternativi disponibili. A questo punto, una volta usciti dal ciclo in questione, se tale contatore non è nullo, la procedura alloca un numero di strutture **IntuiText** uguale al numero di testi alternativi desiderati, non una di più, non una di meno. In questo modo, non solo allochiamo esattamente la quantità di memoria necessaria, evitanto sprechi, ma, posizionando tale operazione fuori ciclo, grazie al contatore, effettuiamo una sola allocazione piuttosto che una serie di allocazioni, una per testo alternativo.

Lo svantaggio di questa tecnica, tuttavia, è il seguente: per evitare di complicare le procedure di fine-programma e per avere una gestione semplice ed efficace della memoria, abbiamo sempre utilizzato il servizio di Intuition **AllocRemember()**. Risulta quindi naturale utilizzarlo anche in questo caso. Questo però implica la necessità di passare alla procedura **SetupItemList()** il puntatore alla struttura **Remember** utilizzato da Intuition come punto di aggancio per la lista dei blocchi di memoria allocati. Ci sono due modi per far ciò: il primo consiste nel passare tale puntatore come parametro all'atto della chiamata, il secondo consiste in un riferimento esterno alla variabile globale all'interno della funzione stessa. Il primo metodo è il più pulito, ma allunga ulteriormente la già lunga lista di parametri da passare alla **SetupItemList()**, costringendoci a rimettere per giunta mano al codice della **BuildMenus()**. Il secondo metodo ci evita pesanti modifiche al codice già scritto, ma ci fa deviare dalla logica a scatola nera che avevamo riservato alle funzioni di servizio. In ogni caso, anche la prima tecnica è concettualmente *sporca*, dato che forza il programmatore a definire un puntatore ad una struttura **Remember** anche nel caso in cui questi avesse deciso di utilizzare un'altra tecnica di allocazione di memoria. In pratica, si lega una funzione di servizio alla logica del programma specifico (ed in particolare della **StartAll()**) perdendo così parte del carattere generale di tale servizio e limitandone l'utilizzo solo in quei programmi che fanno già uso della **AllocRemember()**. Ma, direte voi, cosa importa? L'importante è che il programma funzioni, no? Questo può essere vero se ci limitiamo al singolo programma, ma un programmatore deve, quando scrive un programma, pensare ad ottimizzare tutto il suo lavoro, anche quello futuro. Una opportuna scelta di disegno di un programma, garantisce un riutilizzo di codice ed addirittura degli stessi moduli oggetto, che semplifica sempre di più con l'andar del tempo lo sviluppo di nuovi programmi e la manutenzione dei precedenti. Se poi qualcuno pensa che stia facendo più della filosofia che dei discorsi pratici, libero di pensarlo, ma queste tecniche sono il frutto di anni di programmazione in vari linguaggi, sia miei che di amici e colleghi, ed hanno sempre dimostrato di valere lo sforzo di disciplina effettuato.

Torniamo al codice. Effettuata l'allocazione delle strutture **IntuiText** per i testi alternativi, facciamo partire un secondo ciclo che riempie tali strutture e le associa al campo **SelectFill**. Notare la tecnica che riutilizza il contatore usato nel ciclo precedente per contare i testi alternativi.

Questo, tuttavia non è sufficiente ad attivare il testo secondario. Dato che l'utilizzo di due testi per una singola voce è di fatto una tecnica di evidenziazione della voce selezionata, essa è mutualmente esclusiva con altre tecniche di evidenziazione; è necessario cioè che il campo **Flags** relativo alla voce in questione cotenga **HIGHIMAGE**. Dato però che in precedenza tale campo era stato impostato per mezzo del parametro **itemflags** passato nella lista dei parametri *[parameter list]*, è necessario assicurarsi che questo e solo questo segnalatore sia attivo nel campo **Flags**. Questo è il motivo delle due istruzioni che seguono l'assegnazione del puntatore alla struttura **IntuiText** al campo **SelectFill** (vedi figura 7).

L'ultimo problema da affrontare è il seguente: dato che il testo alternativo può essere più lungo di quello base, è necessario tener conto anche di questa possibilità nel calcolo automatico della larghezza delle liste delle voci e sottovoci, cioè in **itemwidth**.

La possibilità di avere testi base ed alternativi di diversa lunghezza, pone tuttavia un altro problema. Se lanciate il programma e selezionate una voce con doppio testo, vi accorgerete che quando il testo più corto copre quello più lungo, non cancella la parte finale di quest'ultimo che rimane scoperta. Nel nostro caso **ON** diventa **OFF** e poi **ONF**, invece di **ON**. Una soluzione potrebbe essere quella di scrivere due testi della stessa lunghezza fin dall'inizio, evitando così del tutto il problema. Tale soluzione, tuttavia, costringerebbe il programmatore ad una certa cura nello scrivere i testi, e soprattutto a ricordare che *deve* farlo. Tanti sforzi per automatizzare il processo di creazione dei menu ed ora questo... Che fare allora? Beh, chi trova la soluzio-

La scheda tecnica

Continua la nostra carrellata sui comandi dell'*AmigaDos 1.3*. Speriamo di fare in tempo a vederli tutti prima che esca la versione 1.4...

Attenzione

Se si usano i caratteri di sostituzione (wildcard) con i comandi COPY e DELETE è ora permesso di avere nomi di file (con eventuale percorso specificato esplicitamente) più lunghi di 31 caratteri, a differenza di quello che succedeva con la versione 1.2.

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:	COPY
Formato:	COPY [[FROM] <nome>] [TO] <nome> [ALL] [QUIET] [CLONE] [COM] [BUF|BUFFER=<nn>] [DATE] [NOPRO]
Sintassi:	COPY "FROM,TO/A,ALL/S,QUIET/S,CLONE/S,COM/S, BUF=BUFFER/K,DATE/S,NOPRO/S"
Scopo:	Copia uno o più file ad una "device"
Specifiche:	Copia uno più file ad una "device". Se il direttorio destinazione non esiste, esso viene creato (a differenza di quanto succedeva con la versione 1.2). La coppia "" può essere usata per indicare il direttorio corrente. Le opzioni sono: * ALL — copia tutti i file dal direttorio di partenza * QUIET — non visualizza i file man mano che li copia * CLONE — copia anche i commenti, la data e gli attributi * COM — copia anche i commenti associati al file * BUF BUFFER — specifica il numero di buffer da 512 bytes da usare (il default è 200 == 100K) * DATE — copia anche la data di creazione/ultima modifica * NOPRO — non copiare gli attributi (bit di protezione)
Esempio:	COPY "" TO c: ALL copia tutti i file dal direttorio corrente a quello dei comandi

Comando:	BINDDRIVERS
Formato:	BINDDRIVERS
Sintassi:	BINDDRIVERS
Scopo:	Per associare un "device driver" all'hardware
Specifiche:	Viene utilizzato, in genere nella sequenza di partenza, per associare ad un "device driver" che si trova nel direttorio SYS:Expansion, l'hardware addizionale, in modo da far sì che quest'ultimo venga automaticamente configurato grazie ai servizi della "expansion.library".
Esempio:	Se ad esempio una scheda BridgeBoard è disponibile ed in in SYS:Expansion è presente la "Janus.Library", allora la scheda viene automaticamente configurata ed è pronta per il comando "DJHount".

Comando:	DATE
Formato:	DATE [<data>] [<ora>] [TO|VER <file>]
Sintassi:	DATE "DATE,TIME,TO=VER/K"
Scopo:	Visualizza od imposta la data e l'ora corrente
Specifiche:	Se non sono forniti parametri, visualizza la data e l'ora corrente, altrimenti imposta come valori correnti quelli forniti dall'utente. TO o VER servono a reindirizzare la data e l'ora visulalizzate dal comando.
Esempio:	DATE 18:34:22 Imposta come ora corrente 18:34:22

Comando:	CHANGETASKPRI
Formato:	CHANGETASKPRI <priorità> [<processo>]
Sintassi:	CHANGETASKPRI "PRIORITY/A,PROCESS/K"
Scopo:	Cambia la priorità ai processi lanciati da CLI
Specifiche:	Serve a modificare la priorità di un processo CLI. Tutti i processi figli ereditano la stessa priorità di quello genitore. Se non si specifica alcun processo, il comando modifica la priorità di quello corrente. La priorità può andare da -128 (la più bassa) a +127 (la più alta). In genere la priorità di un processo è 0. Si raccomanda di non usare una priorità maggiore di +5 per non entrare in competizione con il sistema stesso. Il comando STATUS permette di conoscere le priorità dei vari processi.
Esempio:	CHANGETASKPRI 3 Porta a 3 la priorità del processo corrente. CHANGETASKPRI -5 PROCESS 4 Porta a -5 la priorità del processo 4.

Comando:	DELETE
Formato:	DELETE <nome>... [ALL] [Q|QUIET]
Sintassi:	DELETE "NAME(S),ALL/S,Q=QUIET/S"
Scopo:	Cancella fino a dieci file o tutti quelli di un direttorio
Specifiche:	Cancella uno o più file. Possono essere specificati fino a 10 file direttamente. Se si utilizzano i caratteri di sostituzione, il numero di file che possono essere cancellati è illimitato. Le opzioni sono: * ALL — cancella tutti i file dal direttorio specificato * QUIET — non visualizza i file man mano che li cancella
Esempio:	DELETE t: ALL Q Cancella tutti i file nel direttorio temporaneo senza visualizzarli. DELETE #? ALL Cancella tutti i file nel direttorio corrente e lista i nomi man mano che li cancella.

```
/*************************************************************************
** 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,  altc = 0;        /* Contatori: voci e testi alternativi      */
   SHORT   itemwidth = 0;   /* Larghezza dell'elemento                  */
   BOOL    anycmd = FALSE;  /* Tiene traccia della presenza di comandi  */
   ITXT   *altname;         /* Puntatore alla lista dei testi alternati */

   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?     */

     /*
     **  Per ora mettiamo il testo alternativo nullo, ma contiamo quante
     **  voci, sottovoci ce l'hanno.
     */
     ilist[i].SelectFill  = NULL;         /* Testo alternativo: nullo     */
     if (itemname[i].alt != NULL) altc++; /* Contiamo quanti sono         */
   }
   ilist[itemnum-1].NextItem = NULL;      /* Ultimo elemento              */

   /*
   **  Adesso allochiamo la memoria per eventuali testi alternativi
   */
   if (altc > 0)
   {
     altname = (ITXT *)AllocRemember(&rememory,  /* Proviamo ad allocare  */
               altc * sizeof(ITXT), MEMF_CLEAR); /* la memoria necessaria */
     /*
     **  Dato che è un peccato perder tutto se non c'è memoria, in questo
     **  caso decidiamo di continuare comunque. Meglio che niente, no?
     */
     if (altname != NULL)
     {
       for (i = 0; i < itemnum; i++)
       {
         if (itemname[i].alt != NULL)  /* Voce con testo alternativo      */
         {
           altc--;    /* Diminuiamo di uno il numero di testi alternativi */
           altname[altc]           = basetext;
           altname[altc].IText     = itemname[i].alt;
           altname[altc].LeftEdge += ((itemflags & CHECKIT)? CHECKWIDTH:0);
           ilist[i].SelectFill     = (APTR)&altname[altc];

           /*
           **  Assicurati che solo HIGHIMAGE sia impostato per questa voce
           */
           ilist[i].Flags         &= ~(HIGHBOX|HIGHCOMP);
           ilist[i].Flags         |= HIGHIMAGE;

           /*
           **  Cerca la larghezza massime tra quelle delle singole voci
           **  Un testo alternativo può infatti essere più lungo del più
           **  lungo testo normale.
           */
           itemwidth  = max( itemwidth, (IntuiTextLength(&altname[altc]) +
           ((itemflags & CHECKIT)  ? CHECKWIDTH : 0)  )); /* Marcatore?  */
         }
       }
     }
   }

   /*
   **  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 7
SetupItemList().

ne può farmela avere in casella via MC-Link (MC2120).

Conclusione

Ormai il nostro scheletro ha già acquisito un aspetto abbastanza *professionale* ed è già in grado di coprire la maggior parte delle funzioni Intuition relative ai menu. Mancano ancora il campo **Mutual Exclude** e le tecniche relative alle voci grafiche e miste. Dobbiamo inoltre risolvere il problema relativo alla allocazione di memoria nella **SetupItemList()** ed al riferimento esterno nella **CloseSafelyWindow()**. Inoltre sarebbe opportuno automatizzare anche la larghezza dei menu tenendo conto di *fonts* differenti da *topaz* 8, eliminando così la costante **CHARWIDTH**. Infine non abbiamo ancora stabilito un meccanismo per inviare messaggi di errore. Infatti, al momento, la **CloseAll()** si limita a chiudere tutto in modo pulito, ma non spiega il perché il programma è terminato in modo anormale (non ha potuto aprire una libreria? Non aveva abbastanza memoria? Non è riuscito ad aprire la finestra principale?).

Questo e molto altro ancora nelle prossime puntate.

Computer Stampanti Monitor

ATARI (DTP CENTER)

1040STFM	999000
1040STE	1249000
SISTEMA DTP	6430000
PCFOLIO	649000

BONDWELL

B200 PORTATILE	1700000

COMMODORE (GARANZIA ITALIANA)

AMIGA 500+20 GIOCHI	749000
AMIGA 2000	1699000

PHILIPS PC PROFESSIONALI

8088 768K 2FD	1100000
8088 768K 1FD HD 20M/CGA	1600000
80286 640K 1FD HD20M/CGA	2100000
80286 640K 1FD HD20M/VGA	2390000
80286 640K 1FD HD45M/VGA	3100000

EASYDATA

BASE286 2FD 1M HD20 MOUSE	1990000
POWER286 2FD 2M HD40M VGA MOUSE	3000000
POWER386 2FD 4M 1HD40M VGA MOUSE	5000000

CITIZEN

120D 80COL/120 CPS	350000
PRODOT9 80COL/300 CPS	850000
PRODOT9X 132COL/300 CPS	1090000
SWIFT24 24 80COL/24 AGHI	649000

NEC

P2PLUS 80COL/24 AGHI	749000

STAR

LC10 80COL/120 CPS	380000
LC10 COLOR	480000
LC24/10 80COL/24 AGHI	699000

PHILIPS

12" MONO	199000
14" MONO	260000
14" VGA MONO	280000
14" COLOR RGB/CGA	499000
14" VGA .39	750000
14" VGA .28	850000

HANTAREX

14" DUAL	280000

NEC

2A 14" VGA	1100000
3D 14" MULTISYNC	1650000

EasyData

TUTTO PER L'INFORMATICA PERSONALE

VIA A.OMODEO 21/29-00179 ROMA

H. 9.30-13.00/15.00-19.30

SABATO COMPRESO-LUNEDI MATTINA CHIUSO

06/7858020

06/7806030

PER ACQUISTI SUPERIORI A L.500.000 IN OMAGGIO DATABANK 8K (VALORE L. 80.000)

Accessori

AMIGA

ESP. 512K	140000
DRIVE	180000
GENLOCK	599000

ATARI

DRIVE	249000
SCART	28000

MS/DOS

SCHEDA VGA (16 BIT - 256K)	290000
SCHEDA S.VGA (16 BIT - 512K)	390000

VASTO CATALOGO SOFTWARE

CTO - LEADER - JSOFT - PD